Contribution à l'élaboration de supports exécutifs exploitant la virtualisation pour le calcul hautes performances

François Diakhaté

N° d'ordre : 4185

Université Bordeaux 1
Laboratoire Bordelais de Recherche en Informatique
CEA/DAM Île de France

THÈSE

pour obtenir le grade de

Docteur
Spécialité : Informatique

au titre de l'école doctorale de Mathématiques et Informatique

présentée et soutenue publiquement le 10 décembre 2010

par Monsieur François DIAKHATÉ

Contribution à l'élaboration de supports exécutifs exploitant la virtualisation pour le calcul hautes performances

Directeur de thèse :	Monsieur	Raymond	NAMYST	
Encadrement CEA :	Monsieur	Marc	PÉRACHE	
Après avis de :	Monsieur	Daniel	HAGIMONT,	Rapporteur
	Monsieur	William	JALBY,	Rapporteur

Devant la commission d'examen formée de :

Monsieur	Daniel	HAGIMONT,	Rapporteur
Monsieur	William	JALBY,	Rapporteur
Monsieur	Jean-François	MÉHAUT,	Président
Monsieur	Raymond	NAMYST,	
Monsieur	Marc	PÉRACHE	

Contribution à l'élaboration de supports exécutifs exploitant la virtualisation pour le calcul hautes performances

Résumé : Ces dernières années, la virtualisation a connu un fort engouement dans les centres de traitement de données. Elle séduit par la grande flexibilité qu'elle apporte, par ses propriétés d'isolation et de tolérance aux pannes ainsi que par sa capacité à exploiter les processeurs multicoeurs. Cependant elle est encore peu utilisée dans les grappes, notamment car son impact sur les performances des applications parallèles est considéré comme prohibitif. Pour pallier ce problème, nous avons conçu un périphérique virtuel de communication permettant l'exécution efficace d'applications parallèles dans une grappe de machines virtuelles. Nous proposons en outre un ensemble de techniques permettant de faciliter le déploiement d'applications virtualisées. Ces fonctionnalités ont été regroupées au sein d'un support exécutif permettant de bénéficier des avantages de la virtualisation de la manière la plus transparente possible pour l'utilisateur, et ce en minimisant l'impact sur les performances.

Mots-clés : Virtualisation, HPC, Support exécutif, Passage de message, Grappe, Réseau rapide

Remerciements

Je remercie tout d'abord mon encadrant CEA, Marc Pérache, pour sa supervision et son soutien durant ces trois années de thèse, ainsi que mon directeur de thèse Raymond Namyst, qui m'a fait confiance dès mon premier stage dans l'équipe Runtime et qui a toujours été une grande source d'inspiration. Je remercie également les rapporteurs, Daniel Hagimont et William Jalby, ainsi que le président du jury Jean-François Méhaut, pour l'intérêt qu'ils ont porté à mon travail ainsi que pour leur commentaires.

Je remercie Pierre Leca, Bruno Scheurer et Hervé Jourdren qui m'ont permis d'effectuer mes recherches dans leurs équipes au CEA, ce dernier s'étant tout particulièrement investi dans le suivi de la thèse.

Durant ces trois années, j'ai été amené à côtoyer les nombreux stagiaires, apprentis, doctorants et post-doctorants qui sont passés par l'espace Nord. Ce fût l'occasion de nombreuses discussions enrichissantes, de débats passionnés sur des sujets divers, ainsi que d'excellents moments de détente que ce soit sur le centre ou à l'extérieur. Je remercie donc, Cédric et Simon, pour m'avoir montré la voie, Gilles, ma "connaissance de bus" :), Gaël, pour nos soirées musique, Jérôme, pour être passé au CEA un soir à 23h, Thomas, pour avoir vécu au rythme des playoffs, Marc, pour sa culture de gamer old-school, Stéphane, pour avoir écrit les pilotes de ma carte graphique, Manu, pour m'avoir fait découvrir les planchettes du Sput, Julien, JB et Matthieu, pour nos sessions basket, Nicolas et son tarif A, Alexandra, parce qu'il ne faut pas contrarier la chef, Jean-Yves, notre brillant entrepreneur, et Sebastien, pour ses cours sur le LHC.

Je remercie par ailleurs Stéphanie, Isabelle, Patrick et Thao, pour leur soutien administratif et logistique aux étudiants de la zone Nord. Je souhaite aussi remercier Aurélien, pour avoir mis en place les noeuds de calcul dont j'avais besoin.

Par ailleurs, lors de mes différents passages à Bordeaux j'ai toujours beaucoup apprécié l'accueil qui m'a été réservé au sein de l'équipe Runtime, ainsi que les échanges avec les – désormais nombreux – membres de l'équipe. Je remercie donc mes ex co-bureau du LABRI, Samuel, François 1, Elisabeth et Sylvain, pour m'avoir fait une place dans un bureau surpeuplé, mon co-bureau de feu le premier bâtiment de l'INRIA, Cédric, pour avoir partagé mon incompréhension de l'espagnol, et ma seconde co-bureau à l'INRIA, Stéphanie, pour avoir facilité mon installation. Je remercie tout particulièrement Nathalie et Stéphanie, pour leur aide à la préparation du pot de thèse, ainsi que François 2, Jérôme et Brice, pour leur aide avant la soutenance. Je remercie enfin tous les membres de l'équipe pour les conseils qu'ils m'ont prodigué ainsi que pour l'ambiance sympathique qu'ils font régner au laboratoire.

Enfin, je souhaite remercier ma famille, pour m'avoir soutenu tout au long de mes études, ainsi que Nathalie, pour ses encouragements sans lesquels les écueils de la thèse auraient parfois pu paraître insurmontables.

Table des matières

Table des figures

Introduction

Ce document présente les travaux réalisés durant ma thèse au CEA/DAM Île de France sous la direction conjointe de Raymond NAMYST et Marc PÉRACHE.

Motivations du travail

La simulation numérique s'est aujourd'hui imposée comme un outil de recherche indispensable dans la majorité des domaines scientifiques, que ce soit au sein des laboratoires universitaires ou dans l'industrie. Cette révolution a conduit à la conception de machines de plus en plus performantes, de façon à augmenter la quantité et la précision des simulations effectuées. Toutefois l'envolée de la puissance des machines dédiées au calcul intensif s'accompagne de l'arrivée de contraintes nouvelles qui sont de plus en plus difficiles à maîtriser.

En effet, du fait de la stagnation de la capacité de traitement séquentiel des processeurs, il a fallu se tourner vers le parallélisme pour augmenter la puissance de calcul disponible pour une simulation : à l'heure actuelle, les machines les plus puissantes au monde sont constituées de centaines de milliers de processeurs mis en réseau. À cette échelle il devient crucial de traiter certains problèmes qui pouvaient être ignorés sur des machines de taille inférieure.

Ainsi, à la difficulté de paralléliser efficacement les codes de calcul en fonction des caractéristiques de la machine s'ajoute la nécessité de prendre en compte les pannes matérielles potentielles. Elles ne peuvent en effet plus être négligées étant donné le nombre de composants mis en jeu. En outre, ces machines parallèles étant de plus en plus coûteuses, il devient important de les mutualiser entre un maximum d'utilisateurs pour rentabiliser aussi bien leur prix d'achat que leur consommation électrique importante. Cela pose des problèmes d'administration nouveaux puisqu'il est difficile de répondre aux besoins d'utilisateurs hétérogènes. Enfin les évolutions matérielles étant toujours plus rapides, il faut être capable de pérenniser des applications dont le temps de développement est supérieur au cycle de vie des machines parallèles.

Toutefois, ces problématiques nouvelles ne sont pas spécifiques au domaine du calcul intensif puisqu'elles sont aussi d'actualité dans les grands centres de traitement de données hébergeant serveurs Web et applications d'entreprise. Ces dernières années, la virtualisation y a été de plus en plus fréquemment utilisée pour faire face à ces difficultés. Elle permet d'exécuter plusieurs machines virtuelles simultanément sur une même machine physique, chaque machine virtuelle étant isolée des autres et disposant de son propre système d'exploitation pour piloter son matériel virtuel dédié.

On dégage ainsi une grande flexibilité puisque chaque utilisateur devient entièrement maître de son environnement d'exécution au sein de sa propre machine virtuelle qu'il peut administrer sans affecter les autres utilisateurs. En outre la virtualisation découple le logiciel du matériel

sous-jacent, ce qui se traduit notamment par la possibilité de migrer les machines virtuelles d'une machine physique à une autre. Cela peut être utilisé à des fins d'équilibrage de charge, de maintenance, ou encore de tolérance aux pannes.

L'utilisation de la virtualisation pourrait donc apporter des réponses intéressantes aux contraintes liées aux machines parallèles de grande taille, mais cette solution était jusqu'ici considérée comme trop pénalisante en terme de performances pour pouvoir être appliquée dans ce cadre.

Objectif de la thèse et contribution

L'objectif de cette thèse était de proposer des solutions nouvelles permettant de profiter simplement et efficacement des avantages apportés par la virtualisation dans le cadre du calcul haute performance (HPC).

Il s'agissait tout d'abord d'analyser les techniques de virtualisation existantes afin de déterminer les raisons qui limitent les performances des applications parallèles lorsqu'elles sont exécutées dans des machines virtuelles. Cette analyse nous a permis de déterminer que, si de récentes avancées ont grandement amélioré l'efficacité de la virtualisation du processeur et de la mémoire, les techniques standard de virtualisation des périphériques de communication restent pénalisantes pour les codes de calcul parallèles. Pour obtenir de bonnes performances, la solution consiste actuellement à laisser les machines virtuelles accéder directement aux périphériques physiques de communication ce qui limite l'intérêt de la virtualisation.

Pour pallier ce problème nous avons conçu un périphérique virtuel de communication permettant l'exécution efficace d'applications parallèles dans un ensemble de machines virtuelles. Ce périphérique se démarque par son interface qui a été pensée de manière à ce que les transferts de données entre machines virtuelles puissent être effectués le plus efficacement possible.

Par ailleurs, nous introduisons un ensemble de techniques permettant de faciliter l'exécution d'applications parallèles dans des machines virtuelles. Nous avons notamment développé une technique permettant d'instancier très rapidement une grappe de machines virtuelles éphémère pour chaque exécution d'application. Nous proposons en outre un accès au système de fichiers hôte depuis les machines virtuelles afin de faciliter la gestion des fichiers d'entrée/sortie nécessaires au fonctionnement de l'application.

Ces fonctionnalités ont été regroupées au sein d'un support exécutif permettant de bénéficier des avantages de la virtualisation de la manière la plus transparente possible pour l'utilisateur, et ce en minimisant l'impact sur les performances. Ce support exécutif est basé sur l'hyperviseur KVM intégré à Linux, et est intégralement implémenté en espace utilisateur, ce qui facilite son déploiement.

Organisation du document

Ce document s'organise en 5 chapitres. Le premier chapitre dresse un état des lieux de l'évolution de l'architecture des machines parallèles, et souligne les difficultés croissantes rencontrées dans l'utilisation de ces machines. Nous présentons les techniques de virtualisation existantes dans un deuxième chapitre. Nous décrivons les bénéfices que la virtualisation peut apporter dans le cadre du HPC et dressons un état de l'art des utilisations qui ont été proposées. Nous

identifions ensuite un certain nombre de défis qui restent à résoudre pour que la virtualisation puisse être largement adoptée. Le troisième chapitre présente nos contributions. Nous y détaillons les solutions que nous apportons afin d'améliorer les performances ainsi que la simplicité de mise en oeuvre de la virtualisation pour l'exécution d'application parallèles. L'implémentation des fonctionnalités proposées est ensuite détaillée dans le quatrième chapitre. Enfin, le cinquième et dernier chapitre est consacré à l'évaluation des performances de notre solution.

Chapitre 1

Architectures parallèles et calcul scientifique

Sommaire

L'objectif de ce chapitre est de présenter un état des lieux du domaine du calcul scientifique haute performance. Partant du constat que les calculateurs parallèles sont aujourd'hui très majoritairement architecturés en grappes de machines multiprocesseurs, nous commençons par étudier les caractéristiques de ces machines dont la complexité augmente d'année en année. Nous traitons ensuite des modèles de programmation parallèle permettant d'écrire des codes de calcul haute performance et nous verrons notamment en quoi ces modèles de programmation peinent à masquer la complexité des architectures de ces calculateurs. Pour finir, nous

nous intéressons aux caractéristiques de codes de calcul scientifique typiques. Cela nous permet de mettre en évidence les difficultés croissantes rencontrées dans l'exploitation efficace des grappes de calcul modernes dans le cadre du calcul haute performance.

1.1 Architecture des grappes de calcul modernes

Les premiers super-calculateurs, apparus dès les années 1960, étaient des machines monolithiques, centralisées autour d'un unique processeur capable de traiter un grand nombre d'instructions par seconde. Pour atteindre de telles performances, ces processeurs étaient basés sur une architecture vectorielle, c'est-à-dire qu'ils étaient capable d'appliquer simultanément la même opération sur un ensemble de données que l'on appelle un vecteur. Néanmoins, à partir des années 1980, il devint de plus en plus difficile de gagner en performance en exploitant cette unique forme de parallélisme. Cela conduisit à l'arrivée de machines massivement parallèles, composées d'un nombre de plus en plus élevé de processeurs indépendants. Ces machines restaient toutefois peu évolutives et étaient le plus souvent constituées de composants conçus spécifiquement pour le calcul haute performance ce qui les rendait coûteuses et donc peu accessibles.

L'arrivée des grappes de calcul, dans les années 1990, a durablement bouleversé le marché des super-calculateurs. Popularisées notamment par le projet Beowulf [1], ces grappes ont une architecture radicalement différente des premiers super-calculateurs. Elles consistent simplement en un ensemble de machines mises en réseau, le tout - d'un point de vue aussi bien logiciel que matériel - étant constitué de composants pris sur l'étagère, c'est-à-dire issus du marché de l'informatique serveur, voire de celui de l'informatique grand public. Les avantages procurés par cette architecture sont nombreux. Tout d'abord les coûts sont très fortement réduits car les composants sont issus d'un marché bien plus volumineux ce qui permet des économies d'échelle importantes. L'effet positif sur les prix est renforcé par l'utilisation de standards ouverts qui permettent d'intensifier la compétition entre les fournisseurs tout en mutualisant certains coûts de recherche et développement. Du côté logiciel, l'utilisation fréquente de programmes libres permet d'une part d'économiser l'achat de licences coûteuses et d'autre part de modifier facilement les systèmes d'exploitation, bibliothèques et autres logiciels pour les adapter aux besoins spécifiques des utilisateurs. En outre, l'architecture de grappe peut être mise en oeuvre à toute échelle et est évolutive ce qui permet à tout un chacun de monter une petite grappe à faible coût tout en gardant la possibilité d'augmenter la puissance de calcul par la suite. Dans le même ordre d'idées, il devient plus facile de personnaliser son calculateur en fonction de ses besoins : certains utilisateurs auront par exemple besoin d'un réseau haute performance pour des applications parallèles fortement couplées, tandis que d'autres pourront se contenter d'un réseau de type Ethernet standard et moins coûteux.

Aujourd'hui les super-calculateurs purement vectoriels ont pratiquement disparu, et les grappes représentent plus de 80% du classement des 500 plus puissantes machines mondiales, le TOP500 [2]. Les 20% restants sont occupés par les machines massivement parallèles qui ont des caractéristiques de plus en plus proches des grappes puisqu'elles se basent souvent sur des composants matériels et logiciels similaires. En effet, les composants pris sur l'étagère utilisés dans les grappes intègrent désormais la plupart des fonctionnalités permettant d'obtenir de hautes performances en calcul scientifique. Les processeurs grand public actuels tels que ceux produits par Intel et AMD exploitent par exemple en standard la vectorisation qui a fait le succès des premiers super-calculateurs grâce à des jeux d'instruction spécifiques.

Cette prédominance de l'architecture de grappe nous conduit à nous y intéresser dans cette

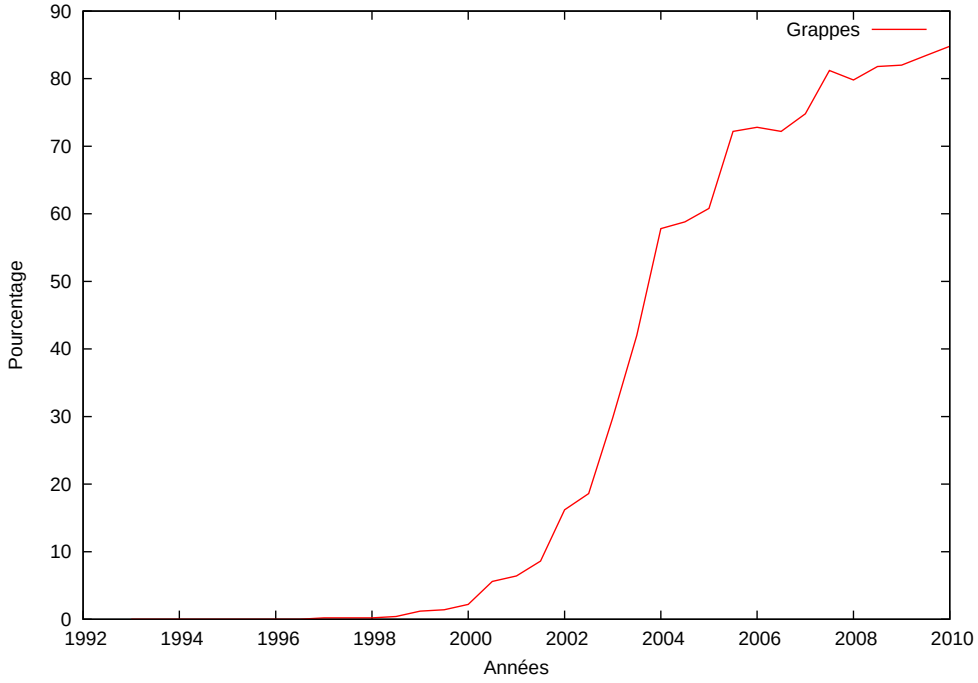


FIGURE 1.1: Pourcentage de grappes parmi les machines du TOP500

thèse et nous allons la décrire plus en détail dans cette section. Nous commençons par analyser le fonctionnement d'un noeud de calcul typique d'une grappe moderne, puis nous décrivons les caractéristiques des réseaux rapides utilisés pour interconnecter ces noeuds.

1.1.1 Des noeuds de calcul de plus en plus parallèles

D'une certaine manière, il est possible d'établir un parallèle entre l'évolution des super-calculateurs et celle des noeuds de calcul constitutifs des grappes. Initialement centralisés autour d'un unique processeur, les noeuds de calcul ont dans un premier temps pu compter sur l'accroissement rapide de la puissance de traitement séquentiel des processeurs pour gagner en efficacité. Le rythme des gains en performance ayant très fortement ralenti depuis les années 2000, du parallélisme supplémentaire a dû être introduit à différents niveaux. D'une part au sein des processeurs, qui intègrent de plus en plus d'unités de calcul indépendantes appelées coeurs, chaque unité de calcul disposant en outre de capacités vectorielles de plus en plus poussées. D'autre part au sein des noeuds, qui contiennent de plus en plus de processeurs ce qui complexifie leur architecture, notamment du point de vue des accès mémoire. Nous décrivons maintenant plus en détail cette tendance et ses conséquences sur l'architecture des noeuds de calcul.

1.1.1.1 Le parallélisme comme passage obligé

Jusque dans les années 2000, la capacité de traitement séquentiel des processeurs, exprimée en nombre d'opérations effectuées par seconde, a augmenté de manière soutenue à un rythme exponentiel. À l'origine de cette progression vertigineuse se trouve l'augmentation de la finesse de gravure des processeurs qui a permis, comme l'a correctement prédit Gordon Moore [1] dès 1965 [3], de multiplier par deux tous les 18 mois le nombre de transistors pouvant être intégrés dans un processeur. Ces progrès en terme de finesse de gravure ont longtemps pu être exploités pour améliorer l'efficacité séquentielle des processeurs, c'est-à-dire leur vitesse de traitement d'une séquence d'instruction, et ce grâce à différents procédés.

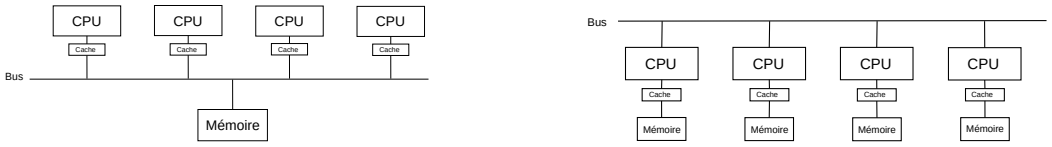
Tout d'abord, la diminution de la taille des transistors libère de l'espace au sein des processeurs pour ajouter de nouveaux circuits permettant d'augmenter le nombre d'instructions traitées par cycle du processeur. La principale méthode employée consiste à essayer d'exécuter simultanément un maximum instructions successives d'un flot d'exécution censé être séquentiel, tout en conservant le résultat attendu. C'est notamment les rôle des pipelines, qui permettent de décomposer le traitement d'une instruction en un ensemble d'étapes successives et d'effectuer la première étape du traitement d'une instruction dès que l'instruction suivante entame la seconde étape. Cela n'est néanmoins possible que si ces instructions n'ont pas de dépendance entre elles. Grâce à l'espace libéré sur les processeurs par l'évolution de la finesse de gravure, de nombreuses techniques matérielles supplémentaires telles que la prédiction de branchement, l'exécution spéculative, ou encore l'exécution *out-of-order* ont pu être implémentées pour réduire l'impact de ces dépendances.

De plus, le raffinement de la gravure des processeurs a longtemps permis d'en augmenter la fréquence d'horloge suivant la même cadence exponentielle. Ainsi jusqu'au milieu des années 2000, il arrivait fréquemment que la loi de Moore soit énoncée par abus de langage comme concernant la fréquence, plutôt que le nombre de transistors. Cette course à la fréquence a atteint son apogée avec l'architecture NetBurst d'Intel qui était pensée expressément pour favoriser une montée rapide de la fréquence d'horloge quitte à dégrader l'efficacité du processeur en terme de nombre d'instructions traitées par cycle (IPC). Outre l'avantage commercial lié à l'affichage d'une fréquence importante, cette architecture aurait dû permettre aux processeurs Intel d'atteindre rapidement des fréquences très importantes [2] et ainsi rentabiliser leur taux d'IPC plus faible.

Enfin, l'espace libéré sur les puces a aussi permis d'augmenter la taille de la mémoire cache des processeurs, qui est très importante pour mitiger le goulot d'étranglement que constitue les accès mémoire. En effet, pour effectuer une instruction, un processeur doit avoir accès à ses opérandes qui sont des données stockées en mémoire. Or, la mémoire centrale étant de nos jours beaucoup moins rapide que le processeur, elle n'est pas capable de lui fournir ces données à un rythme suffisant. Les processeurs intègrent donc des mémoires caches plus petites, mais aussi plus rapides car intégrées sur la même puce, qui vont permettre de stocker les dernières données utilisées dans l'espoir qu'elle servent à nouveau par la suite.

Ces trois leviers, qui ont permis d'améliorer la vitesse d'exécution des programmes sans nécessiter aucun effort d'un point de vue logiciel, ont désormais atteint leurs limites. La quantité de parallélisme d'instruction que l'on peut extraire est limitée et les circuits permettant de s'approcher un peu plus de cette limite deviennent trop complexes. De plus, la fin de la course à la fréquence a été sonnée par l'abandon par Intel de l'architecture NetBurst devant l'impossibilité

1. Gordon Moore est co-fondateur d'Intel et éponyme de la désormais célèbre loi de Moore
2. Un calendrier de lancement d'Intel annonçait des processeurs à 4.4GHz pour 2004 soit bien plus que les processeurs d'aujourd'hui



(a) SMP (b) NUMA

FIGURE 1.2: Examples d'architectures SMP et NUMA

de s'affranchir des problèmes de dissipation thermique, l'énergie consommée augmentant avec le carré de la fréquence. Enfin, passé une certaine taille, l'augmentation de la taille des caches ne permet plus d'améliorer les performances des applications.

Face à cette impasse, la seule manière d'obtenir de nouveaux gains en performance consiste à utiliser plusieurs processeurs en parallèle. La loi de Moore ayant elle toujours cours, les constructeurs utilisent l'espace supplémentaire pour intégrer plusieurs coeurs par puce. C'est donc une véritable révolution qui touche l'ensemble de l'informatique puisque la vitesse d'exécution d'une séquence d'instructions va désormais stagner. Les développeurs doivent maintenant trouver le moyen d'alimenter en parallèle tous les coeurs avec des flots d'instruction différents.

1.1.1.2 Des architectures matérielles de plus en plus complexes

L'augmentation du parallélisme au sein des noeuds de calcul a très fortement complexifié leur architecture, notamment du point de vue des accès mémoire qui deviennent graduellement plus hiérarchiques et plus complexes à maîtriser. On s'éloigne ainsi de plus en plus du modèle des premières machines multiprocesseur symétriques (SMP) dans lesquelles un ensemble de processeurs mono-coeur accédaient à une mémoire centrale par un unique bus (voir Figure 1.2a).

En effet, avec l'augmentation du nombre de processeurs et du nombre de coeurs, ce bus mémoire, qui était déjà un facteur limitant dans le cadre de machines mono-coeur, devient un véritable goulot d'étranglement. Le modèle SMP a donc été abandonné avec par exemple l'introduction des bus HyperTransport chez AMD puis QPI chez Intel. Grâce à ces bus, la mémoire et les processeurs d'une machine vont pouvoir être divisés en un ensemble de noeuds inter-connectés entre eux. Tous les processeurs conservent une vision globale de la mémoire, mais un processeur pourra avoir à effectuer un ou plusieurs sauts au sein du réseau d'interconnexion pour accéder à une mémoire contenue dans un autre noeud. Il en résulte que dans ces machines, l'efficacité d'un accès mémoire dépend de la position de la mémoire par rapport au processeur qui effectue l'accès. On parle donc désormais de machines à accès mémoire non-uniforme (NUMA, voir Figure 1.2b).

En outre, on trouve une structure hiérarchique au sein même des processeurs multicoeurs. Ainsi, ces coeurs ne sont pas totalement indépendants et se partagent plus ou moins de fonctionnalités, avec les avantages et les inconvénients que cela peut présenter. L'exemple le plus notable est celui des mémoires caches. Celles-ci sont généralement organisées en plusieurs niveaux allant du plus rapide et petit, au plus lent et volumineux. Au sein d'un processeur multicoeur, certains niveaux de caches peuvent être privés tandis que d'autres peuvent être partagés entre tout ou partie des coeurs. Des performances optimales ne pourront donc être atteintes que si les coeurs qui partagent un cache travaillent sur des données communes.

Cette tendance à la complexification des architectures matérielles ne semble pas prête de s'arrêter. La tendance est actuellement à l'arrivée de machines hétérogènes contenant des co-processeurs tels que les GPU qui doivent aussi être pris en charge de manière spécifique du côté logiciel. Par ailleurs, avec l'augmentation rapide du nombre de coeurs, il est bien possible que la cohérence mémoire entre les coeurs ne puisse bientôt plus être assurée matériellement...

1.1.2 Réseaux d'interconnexion pour grappes

Peut-être plus encore que l'augmentation de la puissance de calcul fournie par les processeurs pris sur l'étagère, l'amélioration des performances des réseaux a été une des raisons principales derrière le succès des grappes pour le calcul intensif. En effet, les premières grappes étaient basées sur des réseaux standards tels que Ethernet et utilisaient des protocoles de communication inadaptés à cet usage, tels que le protocole Internet (TCP/IP). Elles étaient alors sérieusement pénalisées par le coût de communication entre les noeuds qui empêchait l'exécution à grande échelle d'applications parallèles fortement couplées. C'est l'arrivée, dans les années 1990, de réseaux haute performance dédiés qui a permis aux grappes de devenir une architecture incontournable pour le calcul haute performance. Dans cette section, nous commençons par étudier les raisons qui ont conduit au développement d'interfaces réseau et de protocoles spécifiques. Nous nous intéressons ensuite à leurs caractéristiques afin de comprendre ce qui les rend performants et nous terminons par une présentation des principaux réseaux employés actuellement dans les grappes.

1.1.2.1 Pile TCP/IP standard et grappes de calcul

Le protocole TCP/IP, développé dans les années 1970 et éponyme du réseau Internet est le protocole de communication le plus usité sur les réseaux informatiques. De par son organisation en couches, ce protocole est indépendant du réseau physique sous-jacent et permet d'interconnecter des réseaux hétérogènes. Il est implémenté par tous les principaux systèmes d'exploitation qui proposent une interface standard de type socket pour l'envoi et la réception de données. Son avantage principal est donc sa très grande portabilité d'un point de vue aussi bien matériel que logiciel.

Toutefois, dans le cadre des grappes, ce protocole se révèle peu adapté car il ne permet généralement pas d'atteindre les performances brutes permises par le matériel, notamment en terme de latence. Ce problème d'efficacité trouve ses origines à différents niveaux.

Tout d'abord, le protocole TCP/IP a été conçu pour fonctionner sur des réseaux de natures diverses, aux caractéristiques pouvant être très éloignées des grappes de calcul. Ses capacités, telles que le routage dynamique, la gestion de la congestion et des pannes, ou le contrôle de validité des données sont des fonctionnalités indispensables sur un réseau aussi vaste et distribué que l'Internet mais peu utiles dans le cadre des grappes dont les réseaux sont plutôt statiques et très fiables. Ce sont en revanche autant d'opérations supplémentaires à effectuer sur le chemin critique ce qui impacte négativement la latence des communications.

En outre, la pile TCP/IP étant généralement implémentée dans le noyau, chaque opération va nécessiter un appel système. Cela est vrai aussi bien pour l'envoi ou la réception d'une donnée, que pour simplement en scruter l'arrivée, une opération effectuée très fréquemment. Si le coût des appels système a fortement baissé ces dernières années, la latence des réseaux a suivi la même évolution ce qui fait qu'ils restent pénalisant.

Pour finir, l'interface socket, basée sur l'interface d'accès aux fichiers Unix, n'est pas suffisamment expressive pour permettre de minimiser le coût d'envoi et de réception des données, et ce pour deux raisons principales. D'une part, car elle induit de nombreuses copies intermédiaires, dont certaines doivent être effectuées par le processeur. Les données envoyées sont d'abord copiées dans un tampon intermédiaire situé dans le noyau, pour ensuite être copié par la carte réseau afin de pouvoir être transmises. Le même travail doit être effectué en réception, ce qui rallonge les temps de transfert et a l'effet de bord de polluer les caches des processeurs. Cela est donc doublement pénalisant pour l'efficacité d'un code de calcul parallèle. D'autre part car l'interface socket est généralement synchrone, ce qui ne permet pas de continuer les calculs pendant le transfert des données.

En dépit de son efficacité moindre, le protocole TCP/IP continue à être utilisé dans de nombreuses grappes pour sa simplicité de mise en oeuvre. C'est notamment la seule interface supportée en standard par les systèmes d'exploitation permettant de tirer parti du réseau le plus usité : Ethernet. Cependant, les performances de nombreuses applications parallèles sont fortement impactées par la latence et le débit du réseau, ce qui a conduit au développement d'autres solutions.

1.1.2.2 Caractéristiques principales des réseaux rapides

De nombreux réseaux et protocoles ont été développés dans l'optique de minimiser le surcoût en temps d'exécution lié aux communications inter-noeud. L'analyse précédente des limitations liées aux protocole TCP/IP nous montre déjà que les performances d'un réseau sont liées à des propriétés aussi bien matérielles que logicielles et que l'interface qui est proposée aux développeurs joue un rôle important. Nous étudions donc plus en détail les principales caractéristiques qui donnent aux réseaux rapides leur efficacité supérieure.

Communications en espace utilisateur

Dans la plupart des cas, le système d'exploitation se réserve un droit d'accès exclusif aux périphériques d'une machine. Les programmes en espace utilisateur ne peuvent en tirer partie que par l'intermédiaire de différents appels systèmes implémentés par les pilotes de périphériques. C'est ainsi que les communications réseau standard passent par l'interface socket du système d'exploitation pour chaque envoi ou réception de données, comme vu dans la section 1.1.2.1. Cette contrainte est normalement nécessaire pour des raisons de sécurité – un accès direct permettant de compromettre la machine – ou pour effectuer de manière logicielle un multiplexage entre les besoins de différents programmes utilisant simultanément le périphérique. Pour s'affranchir du coût des appels système sur le chemin critique des communications, les réseaux rapides mettent en place des mécanismes de type OS-bypass permettant d'effectuer les communications en espace utilisateur. Pour cela, ils intègrent un processeur suffisamment sophistiqué pour effectuer matériellement le multiplexage et permettent de sortir les opérations sensibles du chemin critique des communications afin que les transferts de données puissent se faire de manière sécurisée. Typiquement, chaque processus dispose de ses propres files de travaux permettant de piloter la carte réseau, mais celle-ci n'accepte de lire ou d'écrire que dans des zones mémoire préalablement enregistrées. Cet enregistrement se fait toujours par l'intermédiaire du système d'exploitation ce qui permet d'assurer que l'isolation des processus est respectée. En outre, son coût n'impacte pas le chemin critique lorsque les mêmes zones mémoire sont réutilisées pour des communications successives.

Accès direct à la mémoire

Mis à part le surcoût induit par les appels système, nous avons vu que l'efficacité du protocole TCP/IP était impactée par les copies intermédiaires qui doivent être effectuées pour chaque échange de données sur le réseau. Il convient donc, d'une part, de diminuer le nombre de ces copies et d'autre part, de minimiser leur coût. Dans cette optique les périphériques réseaux haute performance sont dotés de la capacité d'accéder directement à la mémoire de la machine, sans que le processeur soit impliqué dans l'opération. On parle de Direct Memory Access (DMA). Le processeur n'étant pas affecté par le transfert, cela évite de polluer sa mémoire cache avec des données inutiles, notamment dans le cas des envois. Cela permet en outre, à l'aide de primitives de communication asynchrones, de continuer à effectuer des calculs pendant que le périphérique réseau s'occupe du transfert. On parle de recouvrement calcul/communication. L'inconvénient du DMA est qu'il introduit un surcoût important sur le chemin critique, chaque transfert nécessitant plusieurs étapes d'initialisation auprès du contrôleur DMA. Cette technique sera donc réservée aux messages de taille suffisamment grande pour que les coûts d'initialisation deviennent négligeables. Dans le cas contraire, il est plus efficace de laisser le processeur se charger de la copie.

Interface de communication

Les possibilités matérielles exposées dans les paragraphes précédents ne peuvent être exploitées efficacement que si les interfaces exposées aux applications sont adaptées. Nous avons déjà vu que les capacités d'OS-bypass permettent d'avoir des interfaces en espace utilisateur pour minimiser les latences et que des interfaces non bloquantes permettent de mieux profiter du potentiel de recouvrement calcul/communication dégagé par les DMA.

Une autre caractéristique des interfaces de communication pour réseaux rapides est qu'elles permettent d'éliminer totalement les copies intermédiaires par une utilisation efficace des DMA. En effet, si les DMA suffisent à éliminer relativement simplement toutes les copies en émission, il est plus complexe de faire disparaître les copies en réception car cela signifie que le périphérique réseau doit être capable de déterminer la destination finale en mémoire du message. Deux paradigmes de communication sont fréquemment utilisés à cet effet : l'envoi/réception de messages et les accès mémoire directs distants (RDMA).

Dans le modèle de communication par envoi/réception de message, émetteurs et récepteurs sont activement impliqués lors de chaque communication. Ils doivent chacun soumettre des requêtes indiquant la localisation en mémoire des données à envoyer ou recevoir, chaque requête étant accompagnée d'un identifiant. Les transferts sur le réseau s'effectuent alors entre les zones mémoire de requêtes d'émission et de réception ayant le même identifiant. La technique du *rendez-vous* est généralement utilisée pour informer l'émetteur de la présence chez le récepteur d'une requête de réception adéquate. Le message est ensuite émis sur le réseau, et un DMA permet de l'écrire directement en mémoire à la réception en fonction de l'identifiant.

Les RDMA permettent quant à eux de directement lire ou écrire dans la mémoire d'une machine distante, chaque machine devant enregistrer au préalable les zones mémoire mises à disposition des autres. Ce paradigme de communication se distingue donc par le fait que hors initialisation, seule l'une des deux machines concernées par l'échange de données y participe activement. Une requête RDMA décrivant à la fois la zone mémoire d'émission et de réception des données, les périphériques de communication du côté émetteur et récepteur vont pouvoir s'échanger les données en faisant intégralement appel à des DMA.

1.1.2.3 Principaux réseaux utilisés dans les grappes

Après avoir exposé les caractéristiques générales des réseaux rapides, nous étudions maintenant individuellement les différentes technologies réseau utilisées dans les grappes actuelles. Comme souvent dans l'informatique, après une période initiale marquée par l'arrivée d'un grand nombre d'acteurs sur le marché, on assiste maintenant à une consolidation du secteur. Citons par exemple des réseaux comme Quadrics ou SCI qui, en dépit d'un certain succès, n'ont pas survécu face à la compétition importante sur ce secteur qui reste relativement confidentiel. Aujourd'hui on peut considérer que seules trois technologies réseaux sont encore utilisées fréquemment dans les grappes : Myrinet, Infiniband et bien sur l'incontournable Ethernet.

Myrinet

Le réseau Myrinet [4], produit par la société Myricom depuis 1994, est l'un des premiers réseaux à avoir été conçu spécifiquement dans l'optique d'interconnecter les noeuds d'une grappe de calcul. La dernière version des périphériques Myrinet, baptisée Myri-10G, propose un débit de 10Gbit/s et se démarque par la capacité de gérer nativement deux protocoles : le protocole MX (pour *Myrinet eXpress*), conçu pour les applications de calcul haute performance et le protocole Ethernet standard afin de s'interfacer à un réseau existant, un réseau de stockage par exemple.

Le protocole MX permet ainsi d'obtenir de très bonnes performances pour le calcul intensif, avec des latences d'environ 2μs et des débits équivalents aux débits théoriques. L'interface de programmation proposée est de haut niveau, basée sur de l'échange de message et l'ensemble des opérations bas-niveau à mettre en oeuvre telles que l'utilisation de DMA, l'enregistrement de la mémoire, le déport d'opérations dans des threads dédiés sont effectuées en interne par la bibliothèque MX. En outre, l'interface d'échange de message proposée se rapproche très fortement de celle de MPI, l'une des interfaces de développement les plus utilisées par les développeurs d'applications parallèles (cf section 1.2.2). De nombreux appels MPI peuvent alors être directement convertis en appels à l'interface MX, ce qui garantit une utilisation optimale du réseau dans ce cadre.

Malgré ces qualités, la part de marché des réseaux Myrinet au sein des machines du TOP500 n'a cessé de diminuer ces dernières années passant de 35% en 2003 à environ un pourcent aujourd'hui. On peut supposer que l'adoption de cette technologie est limitée par le fait que, contrairement à Infiniband et Ethernet, elle est liée à une unique société et qu'elle est peut-être trop dirigée vers le marché de niche que constituent les grappes de calcul.

Infiniband

Infiniband [5] est un standard défini en 1999 par un consortium réunissant des grand noms de l'informatique tels que Compaq, IBM, Hewlett-Packard, Intel, Microsoft et Sun. Infiniband a initialement été conçu comme un bus unifiant toutes les entrées/sorties dans un centre de calcul, permettant de relier directement processeurs et périphériques à un unique réseau. Il devait ainsi remplacer aussi bien les bus utilisés au sein d'une machine pour relier processeurs et périphériques, comme le bus PCI (pour Peripheral Component Interconnect), que les technologies réseau destinées à relier des machines entre elles ou à des systèmes de stockage, comme Ethernet ou Fibre Channel. Cet objectif s'est finalement révélé trop ambitieux et Intel a préféré se retirer du projet pour travailler sur la technologie PCI-Express qui allait s'imposer par la suite comme la génération suivante de bus de périphériques.

C'est en tant que réseau rapide qu'Infiniband a connu un franc succès : partant de zéro en 2005, sa prévalence au sein des grappes du TOP500 atteint maintenant 40%. Ses performances sont en effet excellentes avec des latences proches de la microseconde et des débits pouvant aller de 10 à 40 Gbit/s pour les modèles les plus couramment employés, tout ceci en restant relativement bon marché.

En revanche, si une des forces d'Infiniband est d'être un standard implémenté par de nombreux fabricants, tels que Mellanox, QLogic, Voltaire ou encore Cisco, ce standard ne définit pas d'interface de programmation bas-niveau standard. Il spécifie seulement un ensemble de *verbs*, c'est-à-dire de fonctionnalités, sans en définir la syntaxe d'utilisation. Les interfaces mises au point par les différents fournisseurs n'étant pas compatibles il a fallu attendre l'arrivée du projet OpenFabrics pour disposer d'une interface de programmation bas-niveau portable. L'adoption d'Infiniband a par ailleurs été renforcée par la disponibilité d'interfaces de haut niveau proches des sockets telles que SDP (pour *Socket Direct Protocol*) ou IPoIB (pour *IP over Infiniband*). Cette dernière permet même l'utilisation de sockets standards en encapsulant le protocole TCP/IP dans des trames réseau Infiniband.

Dans le cadre du HPC, ce sont les *verbs* qui sont utilisées afin d'obtenir les meilleures performances. Cette interface a la spécificité d'être extrêmement bas-niveau et d'exposer les nombreuses fonctionnalités de ces cartes réseau. Elles supportent en effet les techniques d'OS-bypass, les RDMA, l'envoi/réception de message, l'utilisation d'un transport fiable ou non, le choix final étant laissé à la charge du programmeur. Cette interface est donc très complexe à maîtriser mais c'est le prix à payer pour exploiter efficacement ce réseau. Les performances obtenues sont en effet bien meilleures que celles obtenues avec une interface standard comme IPoIB.

Ethernet

Bien que ne réunissant pas toutes les caractéristiques d'un réseau haute performance, le réseau Ethernet, introduit dès les années 1970, est le réseau le plus couramment utilisé pour les réseaux locaux. De ce point de vue, les grappes de calcul ne font pas exception à la règle puisque la majorité des grappes du Top500 en sont équipées. C'est en effet une technologie bon marché qui propose, dans sa déclinaison 10Gbit, un débit théorique comparable à ceux des réseaux rapides. En pratique, ce réseau pêche par l'absence d'une pile logicielle standard adaptée au calcul haute performance puisque seul le protocole TCP/IP est supporté en standard. Diverses améliorations matérielles et logicielles ont toutefois été proposées sans avoir rencontré de réel succès pour l'instant.

Du côté matériel, la tendance est à décharger le processeur de certaines tâches coûteuses relatives à la gestion du protocole TCP : certaines cartes réseau haut de gamme qualifiées de TCP Offload Engine (TOE) vont même jusqu'à prendre en charge matériellement la totalité du protocole TCP. Poussant la sophistication plus loin, les périphériques supportant matériellement le protocole iWARP [6, 7], pour Internet Wide Area RDMA Protocol, permettent d'utiliser un protocole RDMA sur TCP sur Ethernet, le tout étant entièrement pris en charge par le périphérique réseau. Si les performances de ces périphériques se rapprochent de celles d'un réseau haute performance, c'est aussi le cas de leur prix. On perd alors l'un des intérêts majeur de la technologie Ethernet, censée être bon marché. De plus les évolutions telles que le TOE ne sont pas toujours bien acceptées par les développeurs de systèmes d'exploitation car elles manquent de flexibilité et s'intègrent mal dans les piles logicielles existante. Ces technologies Ethernet avancées restent donc relativement peu utilisées dans les grappes.

Du côté logiciel, des protocoles alternatifs à TCP ont été proposés afin de se rapprocher des performances brutes du matériel. Citons par exemple GAMMA [8], apparu dès la fin des années 1990, ou plus récemment OpenMX [9]. Ce dernier permet l'utilisation du protocole MX avec des périphériques Ethernet standards et exporte la même interface de type passage de messages à l'utilisateur. Une carte Ethernet standard n'embarquant pas autant d'intelligence qu'une carte Myrinet, certaines opérations doivent être effectuées par le processeur, notamment certaines copies en réception. Toutefois, diverses optimisations comme l'utilisation d'un DMA engine permettent d'en limiter le coût et les performances obtenues sont bien meilleures que celles obtenues en TCP. Néanmoins, ces solutions requièrent des pilotes noyau alternatifs ce qui limite leur adoption pour des raisons de portabilité et de sécurité.

1.2 Programmation parallèle des grappes

Nous l'avons vu, l'architecture matérielle des grappes de calcul se complexifie d'année en année. Que ce soit au sein d'un noeud ou dans les réseaux d'interconnexion il devient de plus en plus difficile de maîtriser toutes les caractéristiques matérielles ayant une incidence sur les performances d'un programme. Il serait donc illusoire de demander un tel degré de connaissance à chaque développeur d'application, d'autant plus que ce sont souvent des scientifiques de domaines divers n'ayant pas nécessairement eu une formation poussée en informatique.

Les grappes doivent donc proposer un environnement logiciel permettant d'abstraire les caractéristiques des machines afin de permettre le développement d'applications portables au sens classique du terme, mais aussi du point de vue des performances, c'est-à-dire qu'une application puisse être efficace sur différentes architectures de grappe.

Malheureusement, il n'existe pas, à l'heure actuelle, de mécanisme efficace permettant de paralléliser automatiquement un code écrit pour fonctionner sur une machine séquentielle. C'est au développeur de spécifier lui-même les opérations qui peuvent être exécutées en parallèle. Pour cela, différents modèles de programmation existent et diffèrent par le degré d'explicitation du parallélisme qui est demandé au programmeur. En effet, un programme parallèle doit générer suffisamment de flots d'exécution pour occuper les différentes unités de calcul d'une machine et doit effectuer les échanges de données et synchronisations nécessaires entre ces flots. Certains modèles de programmation laissent le développeur contrôler directement l'ensemble de ces opérations tandis que d'autres demandent au programmeur de spécifier le parallélisme à l'aide de directives de plus haut niveau pour ensuite générer ces opérations automatiquement.

Nous présentons ici les principaux modèles de programmation utilisés actuellement dans les grappes ainsi que les langages de programmation et bibliothèques correspondants. Nous nous intéressons en particulier à l'efficacité de ces différents modèles en fonction, d'une part des caractéristiques de l'application à paralléliser, et d'autre part de l'architecture matérielle sous-jacente. Nous donnons donc d'abord un aperçu des algorithmes utilisés en simulation numérique qui comptent parmi les plus gros consommateurs de ressources de calcul. Nous détaillons ensuite les modèles de programmation de type mémoire distribuée, ceux de type mémoire partagée et enfin les approches hybrides et analysons leurs avantages et inconvénients respectifs dans ce contexte.

1.2.1 L'exemple de la simulation numérique

La simulation numérique est l'une des applications principales pour les grappes de calcul haute performance. De nombreux phénomènes physiques tels que ceux relevant de la mécanique des

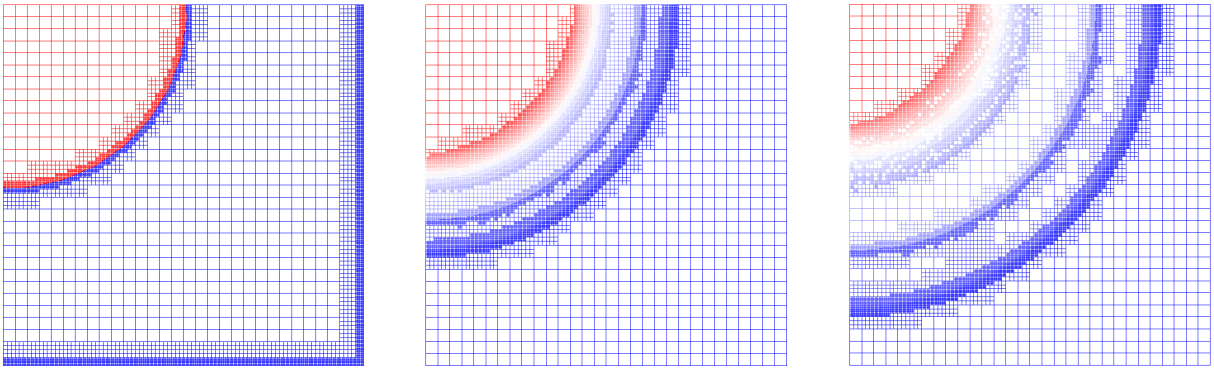


FIGURE 1.3: Évolution dans le temps du raffinement d'un maillage AMR

fluides sont régis par des équations aux dérivées partielles que l'on ne sait pas résoudre analytiquement. Pour contourner ce problème, une discrétisation de l'espace sous forme d'un maillage est souvent utilisée. En effet, en partant des équations différentielles mises en jeu, il est possible de déduire une relation approximant l'évolution sur une courte durée des quantités physiques contenues dans une maille, en fonction des quantités physiques contenues dans les mailles voisines. Cette relation peut donc être appliquée successivement à toutes les mailles du domaine et autant de pas de temps que nécessaire sont effectués pour simuler le phénomène physique sur la durée voulue. Différents types de maillages sont utilisés en fonction du phénomène étudié [10].

Maillages eulériens

Les maillages eulériens sont les plus couramment utilisés. Ces maillages restent fixes dans le temps, et c'est la quantité de matière dans chaque maille qui évolue. On peut alors souvent utiliser des maillages structurés qui se résument à une grille uniforme subdivisant l'espace. Cela permet d'une part d'utiliser une représentation en mémoire sous forme de tableaux, ce qui assure un traitement efficace par les processeurs, et d'autre part de simplifier les relations entre les mailles avec par exemple des méthodes de type direction alternée. L'utilisation d'un maillage structuré pose en revanche problème si les phénomènes physiques ne se répartissent pas uniformément sur le domaine étudié. Par exemple, si la matière a tendance à se concentrer en un point, il est trop coûteux de raffiner l'ensemble de la grille pour améliorer la précision de la simulation autour de ce point.

Maillages adaptatifs

Les maillages adaptatifs permettent de pallier ce problème. L'utilisation d'algorithmes de type AMR (pour *Adaptive Mesh Refinement*) permet en effet de faire varier dynamiquement la précision du maillage et de le raffiner uniquement dans les zones d'intérêt (voir Figure 1.3). Même si l'on perd en efficacité de traitement des mailles puisque leur représentation mémoire devient plus complexe, on gagne en temps de calcul global en en traitant un nombre beaucoup plus faible.

Maillages lagrangiens

Les maillages lagrangiens sont une autre catégorie de maillages fréquemment rencontrés. Ici, chaque maille représente une quantité de matière fixe ce qui fait que le maillage se déplace avec

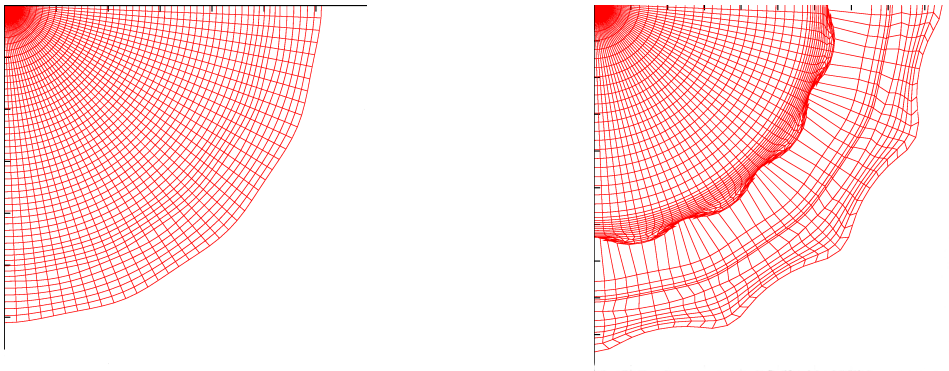


FIGURE 1.4: Évolution dans le temps d'un maillage lagrangien

la matière. Ils permettent notamment de bien suivre l'évolution des interfaces entre plusieurs matières puisque celles-ci sont représentées par les frontières entre les mailles. Ils peuvent en revanche poser des problèmes de stabilité numérique si la taille des mailles varie trop fortement et sont plus complexes à implémenter efficacement que les maillages eulériens.

Conceptuellement, ces algorithmes se prêtent bien à une parallélisation puisque, à chaque pas de temps, le calcul de l'ensemble des mailles peut être effectué en parallèle. En pratique le choix du modèle de programmation influence beaucoup la simplicité et l'efficacité de la parallélisation.

1.2.2 Mémoire distribuée

Dans les modèles de programmation à mémoire distribuée, le programmeur spécifie les tâches devant s'exécuter en parallèle, chaque tâche n'ayant accès qu'à une zone de mémoire privée, inaccessible par les autres tâches. Les interactions entre tâches doivent alors être spécifiées explicitement, à l'aide de primitives permettant d'échanger des données ou de se synchroniser. Parmi les nombreux paradigmes de communication existants, le paradigme de communication par passage de messages est sans conteste le plus populaire à l'heure actuelle.

1.2.2.1 Passage de message

Selon ce modèle, les tâches communiquent entre elles en s'échangeant des messages dont l'envoi et la réception sont explicitement contrôlés par le développeur. Deux types de communications existent.

Communications point-à-point

Les communications point-à-point permettent d'échanger une donnée entre deux tâches. Des primitives d'envoi permettent de spécifier la donnée à envoyer, un destinataire ainsi qu'un identifiant de message. De même, des primitives de réception permettent de spécifier la zone de réception des données ainsi que l'émetteur et l'identifiant de message attendus. Les données sont transférées lorsqu'une requête de réception concorde avec une requête d'émission. Différentes variantes des primitives d'envoi et de réception sont proposées et permettent d'agir par exemple sur le caractère bloquant ou non des communications.

Communications collectives

Les communications collectives impliquent un ensemble de tâches prédéfinies, toutes les tâches devant appeler la même primitive de communication pour que le programme fonctionne. L'objectif de ces primitives de communication est double. Elles permettent d'une part de simplifier la tâche du programmeur en lui évitant d'avoir à implémenter des schémas de communication standards à l'aide de communications point-à-point, et d'autre part de proposer potentiellement une implémentation plus optimisée exploitant mieux les caractéristiques du matériel. Citons par exemple l'opération *broadcast* qui permet de diffuser une donnée d'une tâche vers toutes les autres, ou l'opération *reduce* qui permet de calculer le résultat d'une opération (par exemple une somme) appliquée à un ensemble de valeurs fournies par chacune des tâches.

1.2.2.2 Le standard MPI

Historiquement, le paradigme de communication par passage de messages a été popularisé par PVM (pour *Parallel Virtual Machine*) [11] dont la première version, sortie en 1989, a connu un rapide succès. En effet, les développeurs d'applications parallèles devaient se contenter à cette époque de bibliothèques de communication propriétaires fournies par les constructeurs. À l'inverse PVM est un projet open-source qui trouve son origine dans une collaboration entre plusieurs laboratoires de recherche. Son arrivée a grandement amélioré la situation puisque c'était alors la première bibliothèque permettant d'écrire des applications parallèles portables sur différentes machines, allant même jusqu'à offrir une interopérabilité entre des tâches exécutées sur des machines hétérogènes.

Toutefois, les constructeurs de machines massivement parallèles ayant pris conscience de ce problème de portabilité, un comité composé de chercheurs et d'industriels a été mis en place afin de définir une interface standard pour les communications par passage de message : MPI (pour *Message Passing Interface*) [12]. MPI se distingue de PVM par l'importance plus grande accordée aux performances, en offrant notamment plus de contrôle sur les types de communications employés. En outre, des implémentations diverses de ce standard sont disponibles, certaines étant fournies par les constructeurs des machines et donc optimisées spécifiquement pour tirer partie de leur architecture matérielle. On trouve par ailleurs diverses implémentations libres telles que OpenMPI [13], MPICH [14] et MVAPICH [15] qui tirent très efficacement partie du matériel utilisé couramment dans les grappes, que ce soit pour des communications entre des tâches situées sur un même noeud, ou sur des noeuds différents. Même si MPI n'assure pas l'interopérabilité, il s'est maintenant imposé comme l'interface de référence pour la programmation de machines parallèles au point que certaines interfaces matérielles sont conçues spécifiquement pour pouvoir l'implémenter efficacement (voir section 1.1.2.3).

1.2.2.3 Discussion

La programmation par passage de messages peut être considérée comme une approche bas-niveau mais portable de la programmation parallèle puisque le programmeur garde un total contrôle sur les différentes tâches qui sont exécutées ainsi que sur les communications qui sont effectuées entre elles. Elle encourage une structuration simple du parallélisme ce qui réduit les erreurs de conception et assure naturellement la localité des accès mémoire, chaque tâche travaillant sur ses propres données. Puisque des implémentations efficaces de l'interface MPI existent pour à-peu-près tous les types de grappes, ce paradigme de programmation offre une excellente portabilité des performances ce qui explique que c'est de loin le plus utilisé actuellement.

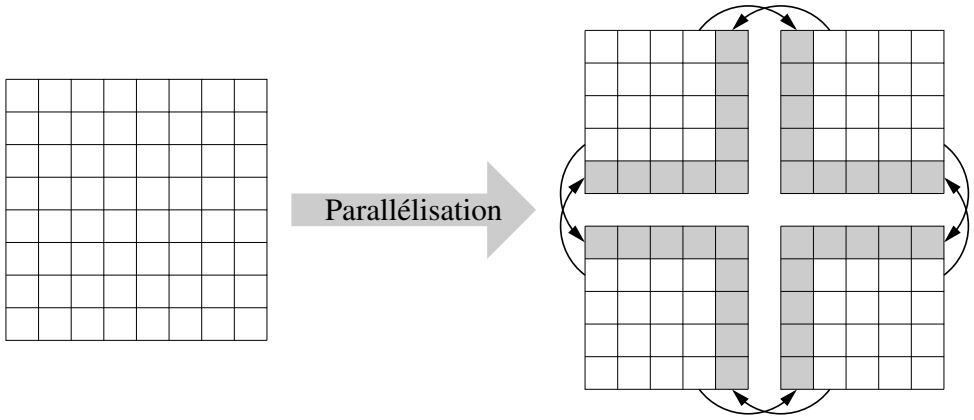


FIGURE 1.5: Parallélisation par décomposition de domaine

La décomposition de domaine est ainsi fréquemment utilisée pour paralléliser les applications de simulation numérique basées sur les maillages que nous avons décrit précédemment [16]. Elle consiste à découper le maillage en autant de sous-domaines que de tâches afin de traiter ces sous-domaines en parallèle. Des mailles fantômes sont alors introduites aux bords de chaque sous-domaine. Elles répliquent les mailles voisines du sous-domaine ce qui permet à chaque tâche de disposer de l'ensemble des données nécessaires à la mise à jour d'un sous-domaine sur un pas de temps. Pour le pas de temps suivant, les mailles fantômes sont mises à jour avec les valeurs calculées dans les sous-domaines voisins grâce à des échanges de messages (voir Figure 1.5).

La difficulté vient néanmoins de la nécessité d'équilibrer la charge entre les processeurs tout en assurant efficacement la distribution des données nécessaires aux calculs. En effet, pour des maillages tels que les maillages adaptatifs, il est nécessaire de prendre en compte l'évolution du maillage au fur et a mesure du calcul afin que chaque processeur traite un nombre équivalent de mailles. Il faut donc détecter ces déséquilibres de charge, repartitionner le maillages en sous-domaines de taille équivalente, et transférer les données vers les tâches qui vont les traiter par la suite.

1.2.3 Mémoire partagée

Dans le modèle de programmation par mémoire partagée, l'ensemble des tâches s'exécutant en parallèle ont une vision globale de la même mémoire. Le programmeur n'a donc pas besoin de spécifier explicitement de communications entre tâches puisque toute donnée écrite en mémoire par une tâche pourra ensuite être directement lue par une autre. D'un point de vue matériel cependant, deux tâches exécutées par deux unités de calcul différentes ne vont généralement pas partager l'ensemble de leur mémoire et l'illusion d'une mémoire partagée doit être maintenue par une combinaison de mécanismes au niveau matériel et logiciel.

Tout d'abord même au sein d'une machine multicoeur, la présence de caches privés aux coeurs nécessite l'utilisation de mécanismes de cohérence de cache pour qu'une donnée modifiée dans le cache d'un coeur soit visible par les autres coeurs. Cette cohérence de cache est effectuée par le matériel mais n'est pas toujours transparente : certains modèles de processeurs nécessitent par exemple l'utilisation de barrières mémoire afin que les opérations effectuées dans le cache d'un coeur soient répercutées dans le bon ordre du point de vue des autres coeurs.

Entre deux noeuds d'une grappe, des mécanismes de mémoire virtuellement partagée doivent

être utilisées (DSM pour *Distributed Shared Memory*). Cette DSM peut être mise en place à bas-niveau, dans le système d'exploitation, ou même dans une couche inférieure, afin de former un système à image unique (SSI pour Single System Image) [17]. Pour les utilisateurs de la grappe tout se passe alors comme s'ils étaient en présence d'une machine unifiée pilotée par un unique système d'exploitation. Une autre solution consiste à mettre en place cette DSM en espace utilisateur, dans le support exécutif. Il est alors possible de tenir compte d'indications fournies par le compilateur ou par le développeur afin de maintenir la cohérence de cache avec un grain plus fin. On obtient alors de meilleures performances, au prix d'une perte de la transparence de la DSM et donc d'une complexification du modèle de programmation.

1.2.3.1 Interfaces

Indépendamment du mécanisme utilisé pour assurer la cohérence mémoire, il existe différentes manières de spécifier le parallélisme, et ce avec des niveaux d'abstraction divers.

Threads

La manière la plus explicite d'exprimer le parallélisme consiste à manipuler directement des threads, ou fils d'exécution, qui se partagent un même espace d'adressage au sein d'un processus. L'interface de programmation pthread (pour POSIX threads) permet ainsi de manipuler des threads de manière portable sur les systèmes d'exploitation respectant les standards POSIX. Le programmeur est maître de la création des threads et surtout de leur synchronisation afin que les dépendances entre les opérations effectuées par les différents threads soient respectées. Diverses primitives de synchronisation sont disponibles à cet effet, telles que les mutex qui permettent de garantir un accès exclusif à une ressource partagée.

Chaque thread dispose de son propre contexte d'exécution (pile d'appel, registres du processeur). Cela permet à l'ordonnanceur d'endormir les threads et d'effectuer des changements de contexte, que ce soit à la suite de l'utilisation de primitives de synchronisation bloquantes, ou à la fin d'un quantum de temps, dans le cadre d'un ordonnancement préemptif. Cette gestion d'un contexte d'exécution complet fait que le coût de création d'un thread n'est pas négligeable. Ce surcoût est exacerbé par le fait que les ordonnanceurs de threads sont généralement implémentés par le système d'exploitation. Les appels à la bibliothèque et les changements de contextes se traduisent donc par des appels système coûteux.

À l'usage, il est difficile de tirer pleinement partie du modèle de programmation par mémoire partagée en manipulant directement des threads. Les interfaces de création de thread sont relativement fastidieuses à utiliser et des problèmes de performance liés au coût des opérations de création et de synchronisation de threads, ainsi qu'à leur ordonnancement font qu'il faut limiter le nombre de threads crées. Pour des applications de calcul scientifique on se contente alors souvent de créer un thread par coeur et de gérer ensuite à la main la répartition du travail entre les threads. On perd ainsi un des avantages de l'approche mémoire partagée qui est de simplifier les problèmes équilibrage de charge par rapport à l'approche mémoire distribuée.

Tâches

Pour pallier ce problème, d'autres interfaces tels que Cilk [18] ou TBB [19] permettent au programmeur de spécifier des tâches indépendamment du contexte dans lequel elles sont ensuite exécutées. La création d'une tâche devient alors bien plus légère que la création d'un thread puisque conceptuellement il suffit de stocker un pointeur vers la fonction à exécuter ainsi

que ses arguments. Ces interfaces proposent en outre une syntaxe simplifiée par rapport à la création de threads. Par exemple, avec Cilk, il suffit de rajouter un mot clé devant l'appel d'une fonction pour que celle-ci soit exécutée dans une nouvelle tâche, et ce avec un faible surcoût : la création d'une tâche ne consomme que 4 fois plus de temps processeur que l'exécution d'un appel de fonction. Les tâches sont ensuite réparties sur les différents processeurs par un mécanisme de vol de travail très efficace et implémenté entièrement en espace utilisateur. La contrepartie est que les primitives de synchronisation proposées sont moins flexibles que celles offertes par les interfaces de manipulation de thread mais ont l'avantage d'engendrer naturellement une structuration simple du parallélisme (de type fork/join). En outre des problèmes de performances peuvent survenir si les tâches effectuent fréquemment des appels système bloquants puisqu'il n'est généralement pas possible de changer de contexte vers une autre tâche. Cependant, les applications de calcul intensif effectuent dans l'ensemble assez peu d'appels système bloquants.

Ces interfaces sont donc bien adaptées aux application de calcul irrégulières qui peuvent ainsi spécifier dynamiquement le parallélisme par un mécanisme simple et efficace. On peut citer tout particulièrement les applications de type diviser pour régner dans lesquelles du parallélisme peut être généré de manière récursive ce qui conduit à un très grand nombre de tâches.

Directives

Enfin, certains langages permettent, directement ou à travers des extensions, de spécifier le côté parallèle de certaines opérations grâce à des directives fournies au compilateur. La création des threads et l'équilibrage de charge entre les processeurs sont ensuite effectués automatiquement. Par exemple HPF (pour *High Performance Fortran*) [20] permet de spécifier des opérations sur des tableaux complets, comme l'addition de deux tableaux, qui pourront ensuite être parallélisées automatiquement. En particulier, la plupart des implémentations existantes fonctionnent sur des architectures à mémoire distribuée et prennent en charge de manière transparente les transferts de données entre les processeurs. OpenMP [21] est une extension pour les langages C/C++ et Fortran qui ajoute un ensemble de directives permettant notamment de spécifier que les itérations d'une boucle peuvent être exécutées en parallèle. Ce type de solution a l'avantage de proposer une approche incrémentale de la parallélisation d'un code séquentiel par l'ajout progressif de directives. Ces directives peuvent en outre être ignorées afin de retrouver le comportement séquentiel initial. Néanmoins, en pratique, des problèmes de passage à l'échelle apparaissent rapidement et, à part pour des applications très simples, il faut souvent repenser la structure de l'application afin de minimiser les portions séquentielles du code. De plus, ces directives ne sont parfois pas assez expressives pour permettre une bonne exploitation des machines multiprocesseur modernes. Par exemple OpenMP ne permet pas de prendre en compte la localité des données dans l'ordonnancement des tâches, ce qui est pénalisant dans le cadre d'architectures NUMA.

1.2.3.2 Discussion

Le modèle de programmation par mémoire partagée décharge le programmeur de la gestion des communications entre les tâches. La mise en place d'une vision globale de la mémoire permet de rompre l'association statique entre une tâche et ses données qui existe dans le modèle de programmation à mémoire distribuée. Il est donc très intéressant pour paralléliser des applications irrégulières pour lesquelles il est difficile de répartir à l'avance les données entre les différentes tâches. Dans le même ordre d'idée, ce modèle de programmation simplifie grandement la création dynamique de tâches et ouvre ainsi la voie à des interfaces de plus haut

niveau pour spécifier le parallélisme. Enfin il permet de minimiser la consommation mémoire en évitant de répliquer les données partagées par plusieurs tâches ce qui devient de plus en plus important avec l'augmentation du nombre de coeurs – et donc de tâches – par noeud.

Néanmoins, en pratique, la programmation par mémoire partagée se révèle complexe à exploiter si l'on souhaite obtenir de bonnes performances. En effet, même si le développeur n'a plus à expliciter les communications entre les tâches, il doit quand même se soucier des communications qui seront mises en oeuvre par le système pour maintenir la cohérence mémoire. Cela peut souvent devenir plus complexe qu'une gestion manuelle des communications car cela nécessite une bonne compréhension de tous les mécanismes de cohérence mémoire mis en jeu et de leurs implications en terme de performance.

Le problème du faux-partage en est un bon exemple. En effet, lorsqu'une tâche modifie une donnée, celle-ci va généralement être placée dans un cache lié au coeur qui a exécuté la tâche. Toute utilisation de la même donnée par une tâche exécutée sur un autre coeur va nécessiter une communication pour assurer la cohérence de cache et sera donc coûteuse. Il faut donc éviter au maximum cette situation, ce qui est rendu difficile par la granularité utilisée par les mécanismes de cohérence de cache. En effet, ceux-ci ne travaillent que par blocs de données de taille variable allant par exemple quelques dizaine d'octets pour les caches des coeurs d'une machine multicoeur à quelque kilo-octets pour une DSM sur une grappe. Ainsi même si l'on a pris soin de faire travailler les tâches sur des données différentes, il faut en outre s'assurer que celles-ci sont disposées de manière suffisamment espacée en mémoire, ce qui n'est pas toujours simple à contrôler.

Dans le même ordre d'idée, la vision d'une mémoire uniformément adressable ne doit pas faire oublier l'importance de la localité des accès mémoire pour les performances. En effet, que ce soit sur une machine de type NUMA, ou plus encore sur une grappe avec une DSM, l'efficacité des accès mémoire varie grandement en fonction de la distance qui sépare le processeur exécutant la tâche du banc mémoire contenant les données. Le développeur doit donc contrôler finement le placement des données et l'ordonnancement des tâches sur les différents processeurs mais les environnements de développement standard ne permettent pas encore de le faire simplement.

Ainsi on constate que, plus les mécanismes nécessaires à la mise en oeuvre de la mémoire partagée ont un coût important, plus il devient complexe d'exploiter efficacement ce modèle de programmation puisque le programmeur rencontre alors de plus en plus de contraintes dissimulées dont l'impact sur les performances est difficile à prendre en compte. De ce fait la programmation par mémoire partagée est surtout utilisée pour exploiter les machines multicoeur modernes qui ont des mécanismes de cohérence de cache efficaces et des effets NUMA faibles. Elle est en revanche peu adaptée pour tirer efficacement partie d'une grappe de grande taille.

1.2.4 Programmation hybride

Comme nous venons de le voir, il est difficile d'exploiter efficacement une grappe en se contentant d'utiliser le modèle de programmation par mémoire partagée. Ce modèle reste néanmoins intéressant car il permet de bien exploiter les machines multicoeur modernes qui constituent les noeuds des grappes. Il permet d'une part de simplifier la parallélisation d'algorithmes irréguliers tout en obtenant une meilleure efficacité, et d'autre part d'économiser de la mémoire en minimisant les duplications de données. Ce gain en mémoire revêt un rôle de plus en plus important puisque, ces dernières années, la quantité moyenne de mémoire embarquée par noeud

a crû moins vite que le nombre de coeurs. Cela implique une diminution de la quantité de mémoire disponible pour chaque coeur, et donc pour chaque tâche. De ce fait, il peut être intéressant de combiner programmation par passage de messages pour la gestion du parallélisme entre les noeuds, et programmation en mémoire partagée pour la gestion du parallélisme intra-noeud. C'est ce que l'on appelle la programmation hybride.

Bien que cette approche soit théoriquement la mieux adaptée pour l'exploitation optimale d'une grappe de calcul moderne, elle est encore peu mise en oeuvre à l'heure actuelle. En effet il n'existe ni interface, ni support exécutif standard prévu à cet effet. Il faut donc combiner des outils conçus pour chacun des deux paradigmes : typiquement une tâche MPI est déployée sur chaque noeud, chaque tâche MPI étant composée d'un ensemble de tâches en mémoire partagée générées par exemple avec OpenMP. Malheureusement, les interfaces des supports exécutifs actuels sont mal adaptées a cette utilisation, et ce pour différentes raisons.

Tout d'abord, les bibliothèques implémentant le standard MPI supportent très mal le mode MPI_THREAD_MULTIPLE qui permet aux primitives de communications MPI d'être appelées simultanément par plusieurs tâches en mémoire partagée. Les performances obtenues lorsque plusieurs tâches communiquent en parallèle sont ainsi très dégradées. En outre, ce mode de fonctionnement étant peu testé, les implémentations standard contiennent encore quelques bugs notamment lorsque des réseaux rapides sont mis en jeu.

En outre, l'ordonnancement des différentes tâches en mémoire partagée rentre en conflit avec la progression des communications MPI ainsi qu'avec les primitives de communication bloquantes car elles sont implémentées par des bibliothèques différentes. Idéalement, la scrutation de la terminaison des communications doit être intégrée à l'ordonnanceur afin de pouvoir efficacement recouvrir les communications par du calcul en exécutant d'autres tâches.

Pour finir, le programmeur doit maîtriser deux interfaces de programmation hétérogènes, qui n'ont pas été pensées pour fonctionner ensemble.

L'approche hybride est donc complexe à mettre en oeuvre et demande un investissement important en temps de développement. Cet investissement se révèle fréquemment peu rentable car, du fait des problèmes exposés ci-dessus, les performances obtenues sont souvent décevantes et même parfois inférieures à celles obtenues par l'approche mémoire distribuée pure. Néanmoins, cette situation est en train de changer du fait l'augmentation rapide du nombre de coeurs embarqués dans les machines et de nombreux travaux de recherche ont actuellement pour but de fournir des supports exécutifs performants pour la programmation hybride[22, 23, 24].

1.3 Des machines complexes à exploiter

Notre étude de l'architecture matérielle des grappes de calcul ainsi que des modèles de programmation dédiés permet déjà de dégager une tendance générale : l'augmentation rapide de la puissance des machines dédiées au calcul haute performance s'accompagne de l'arrivée d'un grand nombre de contraintes. Dans cette section, nous nous penchons plus avant sur les difficultés rencontrées dans l'exploitation des grappes, et ce à différents niveaux. Nous présentons d'abord les obstacles rencontrés par les développeurs qui doivent faire évoluer leurs applications en fonction des architectures matérielles des grappes. Nous passons ensuite aux nouvelles contraintes matérielles à prendre en compte par la pile logicielle de niveau système et terminons par les problèmes de déploiement d'application liés aux utilisations nouvelles des grappes.

1.3.1 Développement parallèle

Comme nous l'avons vu dans la section 1.2, les développeurs doivent paralléliser leurs applications pour exploiter la puissance de calcul des grappes. Ce travail reste extrêmement difficile et les nombreux efforts investis dans la recherche de nouveaux modèles de programmation ainsi que dans le développement d'outils et de bibliothèques adaptés se révèlent encore insuffisants.

Tout d'abord, indépendamment de toute contrainte technique, la nécessité de paralléliser des applications implique une remise en question d'un certain nombre d'acquis. Le meilleur algorithme parallèle n'est par exemple pas nécessairement issu d'une parallélisation du meilleur algorithme séquentiel et un travail de recherche conséquent est nécessaire pour obtenir des solutions efficaces.

En outre, l'écriture d'un programme parallèle reste conditionnée à la prise en compte de nombreuses contraintes liées au matériel que les modèles de programmation présentés précédemment ne permettent pas de masquer. Les efforts déployés pour paralléliser efficacement une application sur une machine donnée peuvent ainsi être réduits à néant lors du passage à une génération suivante de machine. C'est un problème majeur car le matériel tend a évoluer bien plus rapidement que le logiciel : le cycle de vie des machines parallèles est de quelques années seulement, alors que celui de certaines applications critiques se compte en dizaines d'années. Le portage de codes écrit pour des supercalculateurs vectoriels vers les machines parallèles distribuées d'aujourd'hui a par exemple été très coûteux en temps de développement. Aujourd'hui encore, on ne parvient pas à pleinement exploiter les grappes de machines multicoeur mais celles-ci pourraient déjà être remplacées dans un futur proche par des grappes hétérogènes dont la majorité de la puissance de calcul est fournie par des GPUs

Par ailleurs l'ensemble des outils composant l'environnement de développement standard d'un programmeur ne se sont pas encore adaptés à la révolution parallèle. L'un des exemples les plus frappant est celui des outils standard de débuggage qui ne permettent pas d'analyser efficacement des exécutions parallèles à grande échelle. Il en va de même pour les outils de trace qui sont pourtant indispensables pour comprendre les bogues difficiles à reproduire auxquels les développeurs d'applications parallèles sont fréquemment confrontés.

Pour finir, l'utilisation simultanée d'un grand nombre de noeuds de calcul par une application rend probable que l'un des composants matériel mis en jeu tombe en panne au cours de son exécution. Une application dédiée au calcul parallèle à grande échelle doit donc implémenter des mécanismes de tolérance aux pannes pour ne pas gaspiller le temps de calcul déjà consommé en cas de défaillance d'un composant matériel. Ainsi, dans le cas de la technique de protection/reprise, qui est la plus utilisée, l'application doit être capable de sauvegarder sur disque son état complet afin de pouvoir le restaurer par la suite en cas de panne.

Tout cela constitue donc une véritable barrière à l'utilisation massive des grappes de calcul, surtout si l'on considère que leurs utilisateurs potentiels sont souvent des scientifiques de différents domaines, non spécialistes en informatique, qui ont besoin de puissance de calcul mais n'ont pas le temps ou les compétences pour paralléliser efficacement leurs programmes en prenant en compte toutes ces contraintes.

1.3.2 Gestion des contraintes matérielles

En sus du travail supplémentaire induit par la nécessité de paralléliser les applications, l'augmentation de la puissance de calcul des grappes par l'ajout de parallélisme supplémentaire est source de contraintes matérielles nouvelles qui complexifient la gestion des calculateurs. Nous

donnons ci-après quelques exemples de ces problèmes qui nécessitent des solutions novatrices à différent niveaux de la pile logicielle en charge de la gestion de la grappe.

Systèmes d'exploitation

Avec l'augmentation du nombre de coeurs mobilisés pour effectuer des calculs parallèles, il devient impératif de prendre en compte le bruit système qui peut, dans certaines conditions, avoir une influence très importante sur les performances obtenues [25]. On appelle bruit système l'ensemble des traitements relatifs au système d'exploitation que les processeurs d'une machine ont à effectuer, tels que le traitement des interruptions ou l'exécution de différents démons système. Ces traitements asynchrones sont répartis sur les différents coeurs de la machine et « volent » un peu de temps de calcul aux programmes utilisateur qui s'exécutent simultanément. Cependant, comme ils sont généralement très légers, leur impact sur l'exécution d'un programme séquentiel est négligeable. La situation est toute autre dans le cas de programmes parallèles fortement couplés s'exécutant sur des milliers de coeurs. En effet, chaque perturbation d'un coeur par un traitement asynchrone peut potentiellement ralentir l'ensemble des processeurs si ceux-ci ont besoin d'un résultat calculé globalement pour continuer. Ce bruit système ne survenant pas au même moment sur les différents coeurs, son impact sur le temps d'exécution croît avec l'augmentation du nombre de coeurs.

Plus généralement, l'utilisation de systèmes d'exploitation généralistes tels que Linux pose des problèmes croissants. Que ce soit au niveau des politiques d'allocation mémoire, d'ordonnancement ou encore de gestion des entrées/sorties, les algorithmes mis en jeu relèvent de compromis qui ne sont pas toujours optimaux dans le cadre des machines parallèles à grande échelle. Pour tirer les meilleures performances des grappes il est donc fréquemment nécessaire de modifier ces systèmes d'exploitation par l'utilisation de « patchs » voir d'utiliser des systèmes d'exploitation dédiés. Il existe par exemple des systèmes d'exploitation légers qui permettent de minimiser le bruit système et qui laissent plus de contrôle aux applications sur certaines opérations bas-niveau comme les allocations mémoire[26].

Ordonnanceur de travaux

Nous avons vu que les noeuds des grappes ont une architecture hiérarchique. Cela est aussi vrai pour la topologie des réseaux puisque ceux-ci sont souvent organisés selon une cascade de commutateurs réseau formant un *fat tree*. La latence des communications point-à-point entre deux noeuds varie donc suivant le nombre de commutateurs traversés. En outre, pour les grappes de grande taille il devient trop coûteux d'équiper les derniers niveaux de l'arbre de suffisamment de liens pour obtenir une bande passante maximale entre deux moitiés du réseau. On se dirige donc vers des architectures de type « grappe de grappe », contenant un ensemble d'îlots au sein desquels les noeuds peuvent communiquer de manière optimale. Ces îlots sont eux-même interconnectés par un réseau performant mais insuffisant pour permettre à deux îlots complets de communiquer entre eux à pleine vitesse. Il est donc de plus en plus important pour les performances de minimiser la distance sur le réseau des noeuds qui sont utilisés pour l'exécution d'une application parallèle.

De même l'utilisation de noeuds multicoeur et multiprocesseur pose problème. D'un coté, certaines applications très fortement couplées, ou utilisant un modèle de programmation par mémoire partagée, ont besoin de voir leur tâches regroupées sur une même machine. D'un autre coté, d'autres applications gourmandes en mémoire (en quantité ou en bande passante) pourront s'exécuter plus efficacement si leurs tâches sont réparties sur des noeuds différents. Pour

finir, les noeuds de calculs intégrant toujours plus de coeurs, il devient intéressant de regrouper plusieurs applications faiblement parallèles sur un même noeud, afin de minimiser le nombre de noeuds de calcul utilisés et réduire ainsi la consommation électrique.

Toutes ces contraintes sont difficiles à prendre en compte pour les ordonnanceurs de travaux qui ont la charge d'allouer les ressources des grappes pour les applications. Les algorithmes à mettre en place relèvent de compromis entre temps d'exécution, temps d'attente, fragmentation des ressources et nécessitent que l'utilisateur ait une bonne connaissance des caractéristiques de son application pour guider les décisions d'ordonnancement.

1.3.3 Adéquation de l'environnement logiciel

L'augmentation de la taille des grappes rend de plus en plus important de les mutualiser entre un grand nombre d'utilisateurs afin de les rentabiliser, que ce soit au niveau du prix d'achat ou de la consommation électrique. Nous pouvons notamment citer les nombreux projets de grilles qui visent à mettre en commun plusieurs grappes et groupes d'utilisateurs distribués géographiquement sur différents sites. Plus récemment l'engouement pour le *cloud computing* pousse de nombreuses entreprises à développer des offres commerciales permettant de louer de la puissance de calcul à la demande. Ces entreprises disposent en effet de gros centres de calcul très efficaces d'un point de vue énergétique et peuvent donc fournir de la puissance de calcul à faible coût. Pour de nombreux laboratoires, une telle solution peut se révéler plus rentable que l'achat d'une grappe de calcul dédiée.

Cette tendance pose le problème du déploiement rapide d'applications sur des grappes diverses disposant d'environnement logiciels hétérogènes. En effet, même une application écrite de manière portable a souvent des dépendances logicielles, typiquement des bibliothèques, qui ne sont pas simples à satisfaire pour un utilisateur lambda de la grappe. Quant aux administrateurs, ils n'ont pas nécessairement les ressources pour installer et maintenir à jour tous les logiciels requis par les utilisateurs. Ce problème est exacerbé par les incompatibilités qui existent entre les versions de certaines bibliothèques ou compilateurs. Une mise à jour de la pile logicielle installée sur la grappe peut donc empêcher le fonctionnement ultérieur de certaines applications installées par les utilisateurs.

Nous avons par ailleurs vu que des patchs pour le système d'exploitation, voire des systèmes conçus spécialement pour les grappes sont parfois nécessaires pour obtenir les meilleures performances possibles. Ces patchs sont toutefois difficiles à appliquer en pratique car il faut les faire évoluer avec les mises à jour du système et s'assurer qu'ils ne posent pas de problème de sécurité. Quant aux systèmes d'exploitation dédiés, s'ils conviennent à certains utilisateurs pointus désirant les meilleures performances en calcul intensif, d'autres ne peuvent se passer des fonctionnalités standard fournies par les systèmes généralistes.

1.4 Bilan

Ce chapitre a permis de mettre en évidence les difficultés croissantes rencontrées dans l'utilisation des grappes pour le calcul intensif. Les évolutions des architectures matérielles, ainsi que les nouveaux modes d'utilisation des grappes, complexifient le travail des développeurs d'applications parallèles tout comme celui des utilisateurs et administrateurs. Ce constat appelle donc au développement de solutions nouvelles permettant de simplifier l'exploitation des grappes.

Dans cette optique, l'utilisation de machines virtuelles nous semble être une piste particulièrement intéressante. En effet, ces dernières années, la virtualisation un connu un important regain d'intérêt dans les centres de traitement de données qui sont confrontés à des défis similaires. Elle séduit par la grande flexibilité qu'elle apporte, par ses propriétés d'isolation et de tolérance aux pannes ainsi que par sa capacité à tirer partie des processeurs multicoeurs en consolidant les serveurs. Cependant elle est encore peu mise en oeuvre dans le cadre du calcul haute performance, notamment parce que la perte de performance qu'elle induit était jusqu'ici jugée prohibitive. Les avancées récentes dans le domaine de la virtualisation des processeurs généralistes ayant fortement réduit ce coût en performance, nous étudions, dans la suite de ce document, l'utilisation de la virtualisation dans les grappes de calcul.

Chapitre 2

Virtualisation dans les grappes de calcul

Sommaire

Dans ce chapitre, nous nous intéressons à l'utilisation de la virtualisation dans le cadre de l'exécution d'applications de calcul intensif sur grappe. Nous commençons par présenter les différentes formes de virtualisation existantes ainsi que leurs applications respectives. Cette étude nous conduit à nous intéresser plus particulièrement à la virtualisation du matériel qui a connu un important regain d'intérêt ces dernières années. Nous analysons alors en détail les techniques de virtualisation correspondantes afin, notamment, de comprendre dans quelle mesure elles peuvent se répercuter sur les performances d'un code de calcul scientifique. Pour finir, nous détaillons les nombreux avantages que la virtualisation apporte dans le cadre de l'utilisation de grappes et montrons la nécessité de disposer de solutions simples et performantes permettant le déploiement d'applications parallèles virtualisées.

2.1 Tour d'horizon de la virtualisation

À l'heure actuelle, la virtualisation est incontestablement un des sujets en vogue de l'informatique, et le marché des solutions de virtualisation a connu une croissance rapide ces dernières années, soutenu par une forte demande. Cette activité foisonnante autour de la virtualisation a conduit à l'arrivée de nombreux produits nouveaux déclinant le concept de virtualisation de différentes manières ce qui induit une certaine confusion. Nous commençons donc par définir et classifier les nombreuses formes de virtualisation existantes. Nous analysons ensuite l'évolution des solutions de virtualisation d'un point de vue historique afin de mieux comprendre les tenants et aboutissants de l'engouement actuel pour la virtualisation.

2.1.1 Définitions

De manière générale, la virtualisation correspond à l'ajout d'une couche d'abstraction entre une ressource et ses utilisateurs au sens large. L'ajout d'une telle couche intermédiaire permet de dégager une marge de manoeuvre importante dans l'utilisation de la ressource en question. La contrepartie est bien évidemment un surcoût en performance qui varie selon les traitements effectués au sein de cette couche intermédiaire. Les bénéfices apportés par la virtualisation se déclinent selon deux axes principaux.

Tout d'abord, une couche d'abstraction permet d'implémenter une interface d'accès quelconque à la ressource. On peut par exemple choisir l'interface d'une ressource différente afin de donner l'impression que c'est à elle que l'on a à faire, ou définir une interface nouvelle qui sera plus adaptée, que ce soit en terme de portabilité, de simplicité d'utilisation ou de performances.

En outre, tous les accès à la ressource passant par la couche d'abstraction, différents traitements peuvent être mis en oeuvre. Il est ainsi possible de multiplexer la ressource, afin de donner un accès simultané à plusieurs utilisateurs qui auront chacun l'impression d'y avoir un accès exclusif. Par ailleurs on est capable d'intercepter et analyser l'ensemble des accès effectués à des fins de surveillance, de débuggage, d'optimisation ou autre.

La virtualisation englobe donc un grand nombre de techniques et nous présentons ici les principales approches utilisées actuellement. Nous les distinguons en fonction du niveau auquel la couche d'abstraction est mise en oeuvre. Nous nous intéressons d'abord à la virtualisation d'application, puis à la virtualisation de système d'exploitation et enfin à la virtualisation du matériel.

2.1.1.1 Virtualisation d'application

La virtualisation d'application consiste à encapsuler une unique application afin de limiter ses dépendances par rapport au système d'exploitation ou au matériel. L'objectif est généralement d'améliorer la portabilité d'une application et de faciliter son déploiement. On peut distinguer trois catégories de solutions de ce type.

Installateurs virtuels

L'objectif des installateurs virtuels est de permettre d'exécuter une application sur une machine sans avoir à l'installer réellement. La virtualisation s'effectue au niveau du système de fichiers, ou encore de la base de registre sous Windows, afin de faire croire à l'application qu'elle est correctement installée. En réalité tous les accès de l'application à ces ressources sont isolées dans un bac à sable (*sandbox*) et n'affectent pas le système. En incluant toutes les dépendances de l'application dans ce bac à sable, on peut l'exécuter sans installation sur n'importe quelle machine pour peu qu'elle dispose d'un système d'exploitation compatible avec l'application. En outre le système de base reste propre et on évite ainsi les conflits entre applications. Enfin il devient possible de faire de la diffusion en flux d'application (*streaming*) en chargeant les différentes ressources dont elle a besoin au fur et à mesure des accès effectués.

Couches de compatibilité

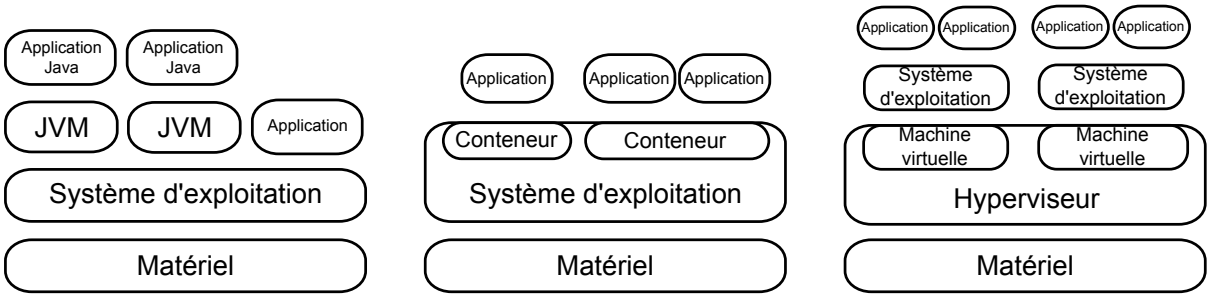
Les couches de compatibilité permettent à une application écrite pour un type ou une version de système d'exploitation spécifique de s'exécuter sur d'autres systèmes d'exploitation incompatibles. Par exemple, Wine permet d'exécuter des applications Windows sur des systèmes de type Unix tels que Linux, MacOS ou Solaris. La technique utilisée est simplement de réimplémenter l'API de Windows en utilisant les fonctionnalités disponibles sur le système d'exploitation hôte.

Machines virtuelles applicatives

Les machines virtuelles applicatives permettent d'isoler complètement l'application du système d'exploitation mais aussi de l'architecture matérielle sous-jacente. Les exemples les plus connus sont les machines virtuelles Java [27] et CLI [1] [28]. Avec ce type de virtualisation, l'application n'est pas compilée pour une architecture matérielle spécifique. La compilation se fait vers un jeu d'instruction synthétique défini par les spécifications de la machine virtuelle. Pour exécuter l'application, il faut donc lancer une machine virtuelle dédiée qui a la charge d'émuler les instructions de l'application en fonction de l'architecture matérielle et du système d'exploitation hôte. Ainsi l'application est compilée une seule fois et peut ensuite être exécutée sur n'importe quelle machine disposant d'une implémentation de la machine virtuelle. Ce modèle améliore en outre la sécurité du système puisque toutes les instructions sont analysées par la machine virtuelle et ne peuvent agir que sur les ressources qui lui sont affectées.

L'émulation des instructions a en revanche un coût très important à l'exécution. Ce coût peut être toutefois fortement réduit par l'utilisation de techniques de compilation à la volée (aussi appelées JIT pour *Just In Time*) qui permettent de compiler dynamiquement le programme pour l'architecture cible au fur et à mesure de son exécution. Cette technique permet même dans

1. Les machines virtuelles CLI sont notamment utilisées pour l'exécution de programmes écrits en C#.



(a) Machines virtuelles applicatives (b) Conteneurs (c) Virtualisation du matériel

FIGURE 2.1: Comparaison de différents types de virtualisation

certains cas de gagner en performance par rapport à une compilation statique puisque le compilateur dispose alors d'informations statistiques supplémentaires, par exemple la fréquence avec laquelle un branchement est pris. Ces informations peuvent être utilisées pour mettre en oeuvre dynamiquement différentes optimisations.

Malgré cela, les machines virtuelles applicatives ne sont pas considérées comme suffisamment efficaces pour les applications de calcul haute performance qui tentent de tirer les meilleures performances des machines. Une réécriture en Java de benchmarks parallèles basés sur des algorithmes standards de calcul scientifique a par exemple mis en évidence une perte de performance allant jusqu'à un facteur six [29]. En outre, la plupart des développeurs sont plus familiers des langages plus proches du matériel tels que le C/C++ et le Fortran et les outils et bibliothèques utilisés dans le domaine sont généralement prévus pour ces langages.

2.1.1.2 Virtualisation de système d'exploitation

La virtualisation de système d'exploitation permet de faire cohabiter sur une même machine plusieurs espaces utilisateurs indépendants et isolés, le tout étant géré par un unique noyau. On appelle ces espaces utilisateur isolés conteneurs ou de zones d'exécution. La virtualisation a lieu au niveau des appels système, par un multiplexage des différents espaces de noms utilisés par le système d'exploitation. Chaque conteneur dispose ainsi de sa propre arborescence de fichiers, de ses propres identifiants d'utilisateurs, de processus, d'interfaces réseau etc... En outre il est possible de partitionner les ressources physiques attribuées à chaque conteneur. On peut ainsi définir des limites sur la quantité mémoire ou le temps processeur qui peuvent être consommés.

Généralement les mécanismes de conteneurs sont implémentés directement au sein de systèmes d'exploitation tels que Linux, FreeBSD, ou OpenSolaris. Leur surcoût en temps d'exécution est très faible puisque la plupart des opérations sont exécutées directement, seul les espaces de noms sont virtualisés. Ils sont donc très intéressants pour partitionner une machine entre plusieurs utilisateurs ou groupes d'utilisateurs indépendants de manière efficace et sécurisée. La limitation est que ces utilisateurs doivent se satisfaire de la même version de système d'exploitation puisque seuls les espaces utilisateur sont virtualisés.

2.1.1.3 Virtualisation du matériel

Les solutions de virtualisation du matériel permettent d'exécuter simultanément plusieurs systèmes d'exploitation existants sur une même machine physique. Elles se basent sur un hypervi-

seur, encore appelé moniteur de machine virtuelle, capable d'émuler simultanément plusieurs machines virtuelles reproduisant l'ensemble de l'interface matérielle de machines physiques existantes. Cela comprend l'ensemble du jeu d'instruction d'un processeur, instructions privilégiées comprises, mais aussi l'interface d'accès aux différents périphériques virtualisés. Ainsi, au sein de chaque machine virtuelle, le système d'exploitation pilote son matériel virtuel dédié, comme il le ferait s'il était exécuté sur une machine physique équivalente. Il est donc totalement isolé des autres machines virtuelles et n'a même pas connaissance du fait qu'il s'exécute sur du matériel virtuel [2].

Si émuler les différentes instructions émises au sein des machines virtuelles est relativement coûteux, de nombreuses optimisations peuvent être mises en oeuvre, notamment dans le cas où le jeu d'instruction du processeur hôte est le même que celui qui est exposé dans la machine virtuelle. Ainsi, récemment, de nombreux travaux ont visé à améliorer la virtualisation du jeu d'instruction x86 qui est utilisé dans la majorité des grappes. Grâce à des techniques de paravirtualisation, ou de virtualisation assistée par le matériel, l'exécution d'un code de calcul séquentiel dans une machine virtuelle induit un surcoût négligeable par rapport à une exécution native. Nous y reviendrons.

La virtualisation du matériel est donc une solution efficace permettant une très grande flexibilité dans l'utilisation d'une machine. Elle permet de partitionner un noeud de calcul en un ensemble de machines indépendantes qui peuvent être entièrement administrées par leurs utilisateurs, jusqu'au choix du système d'exploitation, et ce de manière sécurisée. Cette flexibilité nous semble intéressante dans le cadre de grappes de calcul c'est pourquoi nous nous intéressons à ce type de virtualisation dans la suite de cette thèse.

2.1.2 Historique de la virtualisation du matériel

L'engouement récent pour la virtualisation pourrait laisser à penser que la virtualisation du matériel est une idée qui est apparue ces dernières années seulement, ou tout du moins que sa mise en oeuvre a été permise par l'arrivée de techniques matérielles ou logicielles nouvelles. En réalité, les machines virtuelles étaient utilisées dès les années soixante dans les *mainframes* IBM et mettaient en oeuvre les mêmes principes que les solutions de virtualisation qui sont réapparues ces dernières années. Il est important de bien comprendre les raisons qui ont conduit à l'adoption de cette technologie aussi tôt dans l'histoire de l'informatique, ainsi que celles qui ont mené à son retour en grâce ces dernières années. Nous présentons donc dans cette section un bref historique de la virtualisation, des années soixante à nos jour.

2.1.2.1 Une technique maîtrisée dès les années soixante

L'arrivée de la virtualisation dans les années soixante peut être considérée comme l'une des premières implémentations fonctionnelle d'un système d'exploitation multi-utilisateur à temps partagé. A cette époque, les ordinateurs étaient encore rares, encombrants et très coûteux. Ils ne fonctionnaient en outre qu'en traitement par lots (*batch processing*) : les utilisateurs soumettaient généralement leur programmes par l'insertion de cartes perforées, puis attendaient leur tour dans une file d'attente pour obtenir le résultat.

En 1961 des chercheurs du MIT furent les premier à démontrer la viabilité d'un système d'exploitation à temps partagé avec CTSS (pour *Compatible Time-Sharing System*) [30], donnant l'illu-

2. Certaines heuristiques peuvent néanmoins être utilisées pour détecter l'impact des techniques de virtualisation

sion à plusieurs utilisateurs que leurs programmes s'exécutent simultanément en faisant alterner rapidement le processeur entre les différents programmes. Comme les processeurs de l'époque ne possédaient pas les fonctionnalités nécessaires à l'implémentation d'un tel système, il ne fonctionnait que sur un processeur spécialement modifié pour l'occasion par IBM. CTSS popularisa néanmoins les systèmes à temps partagé en démontrant en pratique leurs nombreux avantages dans le contexte de l'époque. En particulier, ils ouvraient la voie à la pratique d'activités interactives, telles que le débuggage, qui étaient auparavant proscrites car elles monopolisaient des machines dont le temps processeur était précieux.

Suite à ce premier succès, de nombreux projets de systèmes d'exploitation multi-utilisateur à temps partagé ont vu le jour pour les générations suivantes de machines. Citons notamment Multics développé par le MIT et les sociétés General Electrics et Bell Labs, et TSS par IBM. Ces projets se sont malheureusement révélés trop ambitieux et ont accumulé les retards. Cela a conduit Bell Labs à abandonner Multics en 1969 afin de se concentrer sur un projet similaire mais initialement plus simple : Unix. De même TSS fût abandonné par IBM suite à des problèmes de performance et de robustesse.

Développé en parallèle à ces premiers systèmes à temps partagé, CP/CMS connût en revanche un rapide succès [31]. Ce système apportait une solution simple au problème de la gestion d'une machine multi-utilisateur. Il reposait sur l'idée que la meilleure interface à présenter à chaque utilisateur serait l'interface de la machine physique elle-même. L'objectif était donc de donner à chaque utilisateur sa propre machine virtuelle [3], isolée des autres, et aux caractéristiques identiques à la machine physique sous-jacente. Les développeurs de ce système furent les premiers à montrer que cela pouvait être effectué simplement et efficacement grâce au support de la mémoire paginée, nouvellement arrivé dans les processeurs de l'époque. Ainsi, la gestion des machines virtuelles était effectuée par le programme de contrôle, CP, et on pouvait alors installer le système d'exploitation que l'on voulait au sein de chaque machine virtuelle. C'était le rôle de CMS qui était un système d'exploitation simple et mono-utilisateur.

Les raisons du succès de ce système sont proches de celles qui rendent la virtualisation attrayante aujourd'hui. L'architecture choisie permettait de découpler, d'une part, les fonctionnalités bas-niveau du système, très liées au matériel, qui étaient gérées par CP, et d'autre part, les fonctionnalités de plus haut niveau liées au support des applications utilisateurs, qui étaient fournies par CMS. Ces deux composants étaient donc plus simples, pouvaient évoluer indépendamment l'un de l'autre et ils étaient isolés ce qui évitait qu'un bug dans CMS provoque une défaillance de l'ensemble de la machine. En outre, on pouvait faire cohabiter plusieurs systèmes d'exploitation dans différentes machines virtuelles ce qui était important pour garder la compatibilité des applications lors de l'arrivée d'une nouvelle version de système. Enfin, cela permettait aux développeurs système de bénéficier eux aussi de la flexibilité des systèmes à temps partagé puisqu'ils pouvaient alors tester et débugger les systèmes dans des machines virtuelles sans monopoliser la machine physique.

Fortes de ce premier succès, les méthodes de virtualisation ont continué à se développer dans les années qui ont suivi, et de nombreuses techniques, telles que la paravirtualisation (voir section 2.2.1.3), qui sont remises au goût du jour actuellement, étaient déjà mises en oeuvre en production à cette époque.

3. Le terme initialement proposé par les concepteurs du système était pseudo-machine.

2.1.2.2 L'arrivée des ordinateurs personnels

Dès la fin des années 70, l'accès à l'informatique s'est largement démocratisé avec l'arrivée d'ordinateurs de moins en moins encombrants et onéreux. Il devenait donc moins important de pouvoir partager efficacement ces machines entre de nombreux utilisateurs. Cette tendance culmine avec la démocratisation des ordinateurs personnels permettant à tout un chacun de disposer d'un ordinateur chez lui. Suite à l'émergence d'un tel marché de masse, on assiste à une convergence des architectures des serveurs utilisés en entreprise vers celle des ordinateurs personnels ce qui permet de réaliser d'importantes économies d'échelle. Face à une telle abondance de ressources de calcul peu puissantes mais bon marché, la virtualisation est devenue moins attractive, puisqu'il était désormais peu coûteux d'assigner différentes machines à différentes tâches.

2.1.2.3 Un important regain d'intérêt ces dernières années

Depuis les années 2000, on observe un important regain d'intérêt pour la virtualisation. Alors que la part de serveurs virtualisés était négligeable au début de cette décennie, elle atteint maintenant 20% des serveurs nouvellement livrés, et elle pourrait monter à 50% en 2012. Initialement porté par la société VMWare, ce marché est maintenant fortement concurrentiel et a donné naissance à de nombreuses entreprises et produits nouveaux. Cet engouement peut sembler étonnant puisque l'abondance de ressources de calcul qui avait conduit à la mise en retrait de la virtualisation à partir des années 80 est plus que jamais d'actualité. Deux tendances majeures permettent néanmoins de l'expliquer.

L'importance renouvelée du partage des machines entre utilisateurs

Cette tendance est assez intéressante puisqu'elle rejoint celle qui avait conduit au développement initial des machines virtuelles dans les années 60, bien que les contraintes matérielles sous-jacentes soient radicalement opposées. À l'époque les ressources de calcul étaient peu nombreuses et on souhaitait pouvoir les partager entre un grand nombre d'utilisateurs qui n'avaient pas d'autre choix que d'utiliser la même machine. Aujourd'hui, au contraire, la puissance de calcul fournie par un ordinateur personnel standard dépasse souvent les besoins d'un utilisateur. Cette tendance est renforcée par l'omniprésence des architectures multicoeur dont de nombreuses applications ne tirent pas partie car elles ne sont pas parallélisées. Cela amène à vouloir à regrouper plusieurs applications ou utilisateurs sur une même machine de manière à ne pas gaspiller de ressources. Les économies concernent aussi bien la quantité de matériel achetée que les dépenses énergétiques qui constituent une part croissante du coût de revient d'un parc informatique. Ce processus est appelé consolidation de serveur et est la raison principale derrière la résurgence des machines virtuelles.

Bien que la mutualisation des ressources de calcul ait déjà commencé depuis longtemps dans le cadre des nombreux projets de grilles, l'hétérogénéité des environnement logiciels nécessaires aux nombreux utilisateurs ont été un frein à son développement. Ces problèmes sont accentués dans le cadre plus général de la consolidation de serveurs, et la flexibilité offerte par la virtualisation y apporte des solutions intéressantes. Les serveurs Web sont par exemple des candidats typiques à la consolidation. La virtualisation permet d'isoler ces serveurs ce qui améliore leur fiabilité ainsi que leur sécurité puisqu'un crash, ou, respectivement, une attaque d'un des serveurs, n'affectera pas l'ensemble de la machine. En outre, la possibilité de migrer

les machines virtuelles permet d'équilibrer la charge lors des pics d'utilisation. Enfin, la virtualisation permet de faire cohabiter n'importe quelles applications sur une même machine indépendamment de toute incompatibilité logicielle entre elles. On peut penser à des applications qui fonctionnent sur des systèmes d'exploitation différents, ou qui nécessitent des ressources système fixes comme des noms de fichiers ou de segments de mémoire partagés.

Par extension, cette flexibilité a permis le développement de services de type *cloud computing* fournissant de la puissance de calcul bon marché à la demande. Grâce à la virtualisation les utilisateurs restent maîtres de la pile logicielle installée dans leur machine virtuelle tandis que le fournisseur peut optimiser l'utilisation des machines physiques en répartissant la charge et minimiser ainsi les coûts de fonctionnement.

Finalement on retrouve bien l'idée originelle qui avait conduit à l'adoption de la virtualisation pour réaliser les premiers systèmes à temps partagé : pour partager une machine entre plusieurs utilisateurs, l'interface la plus propre à fournir à chaque utilisateur est celle de la machine elle même.

La complexification des logiciels et l'évolution rapide des architectures matérielles

Outre le besoin de partager simplement des machines entre de nombreux utilisateurs, l'adoption de la virtualisation est plus généralement favorisée par la difficulté croissante pour les logiciels de suivre les évolutions matérielles.

En effet, on constate une évolution de plus en plus rapide des architectures matérielles, qui ont tendance à devenir plus difficiles à exploiter efficacement. Ces dernières années ont ainsi vu l'arrivée de processeurs multicoeur et d'accès mémoire de type NUMA jusque dans les machines de bureau. Tout indique que cette tendance va s'intensifier puisqu'on annonce déjà, pour les années à venir, l'arrivée de processeurs massivement multicoeur n'assurant pas matériellement la cohérence de cache [32].

Face à cela, la quantité de logiciel existant continue de croître en volume comme en complexité. Cela implique une inertie importante face à ces évolutions matérielles qui ne sont pas transparentes pour le logiciel. Cette tendance affecte par ailleurs les applications autant que les logiciels de niveau système. En effet, nous avons déjà vu que les applications doivent être parallélisées pour tirer partie des nouvelles architectures parallèles, et que les méthodes de parallélisation sont encore trop dépendantes du matériel visé. Les systèmes d'exploitation sont quant à eux toujours plus complets notamment au niveau des interfaces avec les applications qui ne cessent de s'étoffer et qui doivent être maintenues indéfiniment afin d'assurer la pérennité des applications. Aussi, les systèmes existants sont lents à s'adapter aux évolutions matérielles, et il est aujourd'hui très difficile de réécrire un système d'exploitation en partant de zéro.

La virtualisation apporte une solution intéressante à ce problème car elle permet de masquer certaines évolutions matérielles à des fins de portabilité ou d'efficacité. Nous avons déjà évoqué le cas des machines multicoeur. Puisque de nombreuses applications ne sont pas capables d'en tirer parti, la virtualisation permet de les subdiviser en un ensemble de machines virtuelles mono-coeur qui peuvent être utilisées efficacement par différents utilisateurs. Cette technique est aussi très utilisée dans l'embarqué où les processeurs multicoeur font aussi leur apparition. L'objectif est de dédier un coeur à l'exécution un système d'exploitation temps-réel monocoeur, en charge de piloter le matériel, et d'utiliser les autres coeurs pour la gestion des applications haut niveau avec un système d'exploitation généraliste.

Une idée similaire peut être appliquée dans le cadre du calcul haute performance puisque de nombreux systèmes d'exploitation spécialisés ont été proposés pour améliorer les perfor-

mances des applications de calcul scientifique parallèle sur les architectures matérielles d'aujourd'hui. Ces systèmes étant généralement minimalistes, ils ne fournissent pas toutes les fonctionnalités nécessaires à certaines applications. La virtualisation permet donc de faire cohabiter un système d'exploitation généraliste pour faire fonctionner les applications qui en ont besoin.

Enfin certains détails matériels peuvent être masqués par l'hyperviseur lui-même, afin d'offrir la vision d'une machine plus simple à exploiter. Comme à l'époque de CP/CMS, l'idée est de découpler les fonctionnalités bas-niveau de l'OS très liées au matériel des fonctionnalités plus haut niveau plus liées au support des applications utilisateurs. Par exemple, l'hyperviseur Disco [33] avait été proposé à la fin des années 90 afin d'optimiser les accès mémoire sur machine NUMA, à une époque ou de nombreux systèmes d'exploitation n'étaient pas encore capables de les prendre en compte. Il permettait notamment de migrer ou répliquer les pages mémoire en fonction des processeurs qui y accédaient, et ce de manière transparente vis-à-vis des systèmes d'exploitation dans les machines virtuelles qui avaient la vision d'une mémoire uniforme.

2.2 Techniques de virtualisation du matériel

Nous détaillons à présent les principales techniques de virtualisation du matériel existantes, afin de comprendre leurs avantages et inconvénients respectifs, ainsi que leur impact sur l'exécution de codes de calcul parallèles. Nous nous intéressons plus particulièrement à la virtualisation de machines basées sur l'architecture x86, qui est présente dans la majorité des grappes actuelles. Néanmoins les principes évoqués ici s'appliquent à la plupart des architectures matérielles d'aujourd'hui. Chaque sous-section présente les méthodes de virtualisation relatives à l'un des principaux composants d'une machine : tout d'abord le processeur, puis la mémoire et enfin les périphériques.

2.2.1 Virtualisation du processeur

Le jeu d'instruction x86, initialement introduit en 1978, et étendu par la suite à chaque nouvelle génération de processeur, n'a pas été conçu pour être virtualisé simplement. Les premiers processeurs utilisant ce jeu d'instruction étaient principalement destinés à l'embarqué et aux premières machines de bureau qui étaient trop peu puissantes pour qu'il soit intéressant de les virtualiser. Aujourd'hui en revanche, les processeurs x86 sont omniprésents, tout particulièrement dans les machines ayant d'importantes capacités de calcul généraliste. Du fait de la résurgence de la virtualisation ces dernières années, de nombreux efforts ont été tournés vers la virtualisation efficace de ce jeu d'instruction. Dans cette section, nous commençons par présenter les raisons qui rendent difficile la virtualisation des processeurs x86 par rapport aux processeurs des *mainframes* IBM que l'on savait virtualiser dès les années 60. Nous analysons ensuite les techniques qui peuvent être mises en oeuvre pour contourner ces problèmes, à savoir la paravirtualisation, la translation binaire et la virtualisation assistée par le matériel.

Note : L'une des caractéristiques du jeu d'instruction x86 est d'avoir cherché à assurer au maximum la compatibilité ascendante des programmes. De ce fait, certains des modes d'exécution disponibles, tels que le mode réel, ne sont plus utilisés aujourd'hui. Afin de simplifier notre propos, nous ne nous intéresserons par la suite qu'au mode protégé dans ses déclinaisons 32 bits et 64 bits.

2.2.1.1 Problématique liée au jeu d'instruction x86

Dans les années soixante-dix, Goldberg définit les conditions pour qu'un jeu d'instruction soit virtualisable efficacement [34, 35]. Un jeu d'instruction est dit « efficacement virtualisable » si l'immense majorité des instructions envoyées au processeur virtuel peut être exécutée directement sur le processeur physique, c'est-à-dire sans intervention de l'hyperviseur. Sont donc exclues toutes les techniques d'émulation et de compilation à la volée, qui nécessitent que chaque instruction soit analysée et traduite par l'hyperviseur avant d'être exécutée.

Ces critères permettent en revanche de déterminer si un hyperviseur peut être construit par une technique dite de « déprivilégisation », aussi appelée *trap and emulate*. Le principe général est d'exécuter directement l'ensemble des instructions de la machine virtuelle dans le mode le moins privilégié du processeur. Lorsque des instructions privilégiées sont utilisées dans la machine virtuelle, généralement par le système d'exploitation, une interruption logicielle est déclenchée et va permettre à l'hyperviseur de reprendre la main. Il peut alors analyser l'instruction fautive et émuler le résultat attendu par la machine virtuelle.

Dans ce cadre, trois conditions doivent être remplies pour qu'un jeu d'instruction soit virtualisable :

1. Les instructions non-privilégiées doivent avoir la même forme quel que soit le niveau de privilège courant. Par exemple, certains processeurs utilisent un bit supplémentaire pour l'adressage des données lorsqu'ils sont en mode privilégié. Ce bit étant tout simplement ignoré en mode non-privilégié, cela écarte la possibilité d'employer toute technique de déprivilégisation.
2. L'accès d'une machine virtuelle à la mémoire doit pouvoir être restreint à ses propres données, afin de protéger l'hyperviseur et les autres machines virtuelles. Un mécanisme tel que la translation d'adresse peut être utilisé à cet effet.
3. Toutes les instruction dites « sensibles » doivent être privilégiées, afin que leur utilisation dans une machine virtuelle puisse être interceptée par l'hyperviseur grâce à l'interruption logicielle déclenchée.

On appelle instruction sensible toute instruction dont l'exécution directe par une machine virtuelle pourrait compromettre l'isolation assurée par l'hyperviseur. Goldberg distingue quatre classes d'instructions sensibles.

1. Les instructions touchant au mode d'exécution de la machine. Cela inclut par exemple les instructions permettant de changer le niveau de privilège courant, ou d'activer l'utilisation de la mémoire paginée.
2. Les instructions capables de lire ou écrire dans certains registres ou emplacement mémoire sensibles tels que ceux qui contrôlent les vecteurs d'interruption ou les compteurs de cycles.
3. Les instructions permettant d'accéder aux systèmes d'adressage mémoire et autres mécanismes de protection utilisés par l'hyperviseur pour isoler une machine virtuelle des données qui ne lui appartiennent pas.
4. Les instructions d'entrée/sortie.

Les machines IBM 360/67 satisfaisaient, par chance, à l'ensemble de ces conditions, ce qui a permis de baser l'hyperviseur CP sur la technique du *trap and emulate*. Nous allons voir que ce n'est pas le cas du jeu d'instruction x86.

Les processeurs x86 disposent de quatre niveaux de privilèges (appelés CPL pour *Current Privilege Level*). Les applications s'exécutent généralement au niveau 3, le moins privilégié, tandis que le système d'exploitation s'exécute au niveau 0, le plus privilégié.

Le premier critère de Goldberg est rempli car les instructions sont les mêmes quel que soit le CPL. Simplement, les instructions privilégiées déclenchent une interruption logicielle si le CPL n'est pas 0.

Les processeurs x86 supportent à la fois l'adressage mémoire par segmentation et par pagination. L'accès à chaque segment ou page peut être restreint aux programmes s'exécutant avec un niveau de privilège suffisant. Le jeu d'instruction x86 satisfait donc au deuxième critère de Goldberg.

En revanche, Robin et Irvine [36] ont montré que le 3ème critère de Goldberg est violé par 17 instructions sensibles [4] qui ne sont pas privilégiées. Ces instructions sensibles se répartissent au sein des classes 2 et 3 identifiées par Goldberg. Nous en donnons deux exemples :

– L'instruction POPF permet de modifier le registre EFLAGS du processeur. Ce registre est composé d'un ensemble de drapeaux influant le fonctionnement du processeur. Certains de ces drapeaux sont sensibles, notamment le drapeau IF, qui permet de désactiver les interruptions. De ce fait, ils ne peuvent être modifiés que si le niveau de privilège courant est suffisant. Cependant, dans le cas contraire, la tentative de modification est simplement ignorée et ne déclenche pas d'interruption logicielle. L'instruction POPF est donc une instruction sensible de classe 2 qui n'est pas privilégiée.
– L'instruction MOV permet, quel que soit le niveau de privilège courant, de copier le contenu registre du CS dans un registre général et donc de lire son contenu. Or, le registre CS contient le sélecteur de segment de code. En particulier, les deux premiers bits de ce registre contiennent le niveau de privilège courant. Un système d'exploitation qui exécute cette instruction verra qu'il ne s'exécute pas au niveau de privilège 0 mais au niveau de privilège inférieur mis en place par l'hyperviseur. L'instruction MOV est donc une instruction sensible de classe 3 qui n'est pas privilégiée.

De ce fait, l'architecture x86 ne peut être virtualisée en se contentant d'appliquer la technique du « *trap and emulate* ». Cela n'empêche pas d'utiliser des techniques alternatives, comme nous allons le voir dans les sections suivantes.

2.2.1.2 Translation binaire

La translation binaire s'écarte du cadre de la virtualisation efficace définit par Goldberg car elle ne permet pas d'exécuter les instructions du processeur virtuel directement sur le processeur hôte. Le principe est au contraire de les analyser afin de les traduire en une séquence d'instructions émulant le comportement attendu. C'est donc une technique très générale puisque le jeu d'instruction émulé peut différer de celui du processeur hôte. À priori, le coût d'analyse et de traduction des instructions est important, mais il peut être fortement réduit par diverses optimisations, notamment dans le cas ou la traduction à lieu entre deux jeux d'instructions identiques.

Caches de traduction

Traduire les instructions une à une, à la manière d'un interpréteur, aurait un coût prohibitif. Des caches de traduction sont donc utilisés afin de ne traduire le code qu'une seule fois. Ainsi, le traducteur n'est appelé que lorsque le pointeur d'instruction du processeur virtuel arrive à une adresse mémoire qui n'a pas encore été analysée. Les instructions suivantes sont alors

4. sur environ 250 instructions supportées par les processeurs Pentium disponibles au moment de l'étude

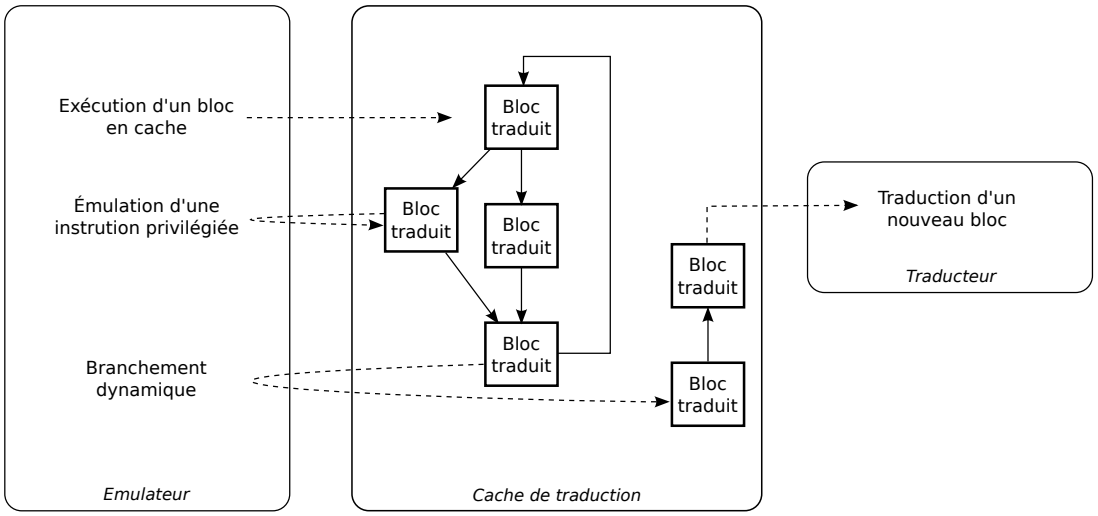


FIGURE 2.2: Translation binaire

traduites et stockées dans un bloc jusqu'à ce que le flot d'instruction soit interrompu, typiquement à cause d'un branchement. Ce bloc est ensuite gardé en cache et peut être retrouvé par une table de hachage qui l'associe avec l'adresse mémoire des instructions correspondantes dans la machine virtuelle.

Si le processeur virtuel utilise le même jeu d'instruction que l'hôte, la traduction est très efficace puisque la plupart des instructions peuvent être traduites à l'identique. Les principales instructions qui doivent faire l'objet d'un traitement particulier sont :

– **Les instructions qui font référence aux adresses mémoire du code**. Puisque le code qui est réellement exécuté n'est pas situé à la même adresse que le code d'origine, il faut modifier ces instructions en conséquence. En particulier, l'adresse des branchements dynamiques ne peut être connue au moment de la génération des blocs de code. Ces instructions doivent donc être remplacées par une recherche du bloc de destination dans la table de hachage.
– **Les instruction sensibles**. Certaines opérations peuvent être émulées directement au sein du bloc de code traduit. C'est notamment le cas d'instructions dont l'émulation consiste simplement à modifier une variable indiquant l'état du processeur virtuel. D'autres opérations plus complexes, comme celles touchant aux mécanismes de protection mémoire, nécessitent de faire appel à la couche d'émulation. Cet appel peut être directement inséré dans le code traduit, ou être effectué indirectement suite au déclenchement d'une interruption logicielle.

Adaptation du code traduit

Nous avons évoqué, dans la section 2.1.1.1, les techniques de compilation à la volée permettant d'améliorer les performances des machines virtuelles Java. La translation binaire se prête aussi à ce type d'optimisations puisque le code traduit peut être adapté en fonction d'informations statistiques recueillies à l'exécution.

Par exemple, dans le cas de l'architecture x86, les mises à jour de la table des pages sont effectuées en écrivant directement dans les zones mémoire correspondantes (voir section 2.2.2.1). Comme les adresses de ces zones mémoire ne sont connues qu'à l'exécution, il n'est pas possible de savoir à l'avance quelles instructions modifient la table des pages. La solution généralement retenue est de protéger en écriture les zones mémoire déclarées comme table des pages

par la machine virtuelle, afin que les modifications puissent être détectées grâce aux interruptions logicielles qui sont déclenchées.

L'adaptation de code permet d'éviter une partie de ces interruptions logicielles très coûteuses en remplaçant les instructions fautives par un appel direct à un interpréteur qui pourra les émuler efficacement.

Exécution directe

Enfin, il existe des hyperviseurs hybrides qui utilisent à la fois la translation binaire et la déprivilégisation. L'idée est d'exécuter directement le code de la machine virtuelle par la technique de déprivilégisation quand l'état du processeur virtuel le permet, et de se rabattre sur la translation binaire dans le cas contraire. Cependant, du fait des contraintes exposées dans la section 2.2.1.1, cette technique demande d'effectuer un compromis entre vitesse d'exécution et fidélité de la virtualisation.

– Le code non privilégié (niveau 3) de la machine virtuelle peut généralement être exécuté directement sur l'hôte bien que certaines instructions rarement utilisées renvoient alors des valeurs incorrectes. De plus certaines opérations, comme la lecture du compteur de cycles, ne peuvent être virtualisées et renvoient la valeur correspondant au processeur hôte.
– La plupart des systèmes d'exploitation modernes n'utilisent que peu les instructions sensibles problématiques évoqués dans la section 2.2.1.1. Si l'on se restreint à l'utilisation de tels systèmes, la mode d'exécution privilégié peut en grande partie être virtualisé par *trap and emulate*. Par exemple KQEMU n'utilise la translation binaire que lorsque les interruptions sont désactivées et parvient à virtualiser correctement certaines versions de Linux et Windows. Une autre solution, utilisée par VirtualBox, consiste à analyser le code privilégié avant de l'exécuter et à remplacer directement dans le code les instructions problématiques par un appel à l'interpréteur.

Toutes ces optimisations permettent aux hyperviseurs utilisant la translation binaire d'obtenir des performances quasi-natives. Elles illustrent en outre l'un des avantages principaux de la translation binaire qui, en tant que technique purement logicielle, est flexible est évolutive.

Cependant, elles complexifient fortement la tâche du traducteur, qui n'est déjà pas simple. Les risques de bugs sont donc considérables ce qui pose des problèmes de sécurité, que ce soit au sein d'une machine virtuelle, si les mécanismes de protections mémoire ne sont pas correctement émulés, ou pour la machine hôte si une erreur du traducteur conduit à générer un code qui accède à des ressources hors de la machine virtuelle [37].

Enfin l'intérêt pour les techniques de translation binaire a diminué avec l'adoption progressive des extensions 64 bits du jeu d'instruction x86. En effet, en mode 32 bits, le traducteur et le code généré peuvent tout deux être exécutés dans le même espace d'adressage. C'est une des raisons derrière l'efficacité de la translation binaire puisque cela permet d'éviter de changer de contexte à chaque fois qu'il faut faire appel au traducteur. Pour cela, le traducteur doit protéger ses données en les plaçant dans un segment qui n'est pas accessible lorsque le code traduit s'exécute. Cette technique n'est plus possible en mode 64 bits car le support de la segmentation n'y est plus complètement assuré.

2.2.1.3 Paravirtualisation

La paravirtualisation est une autre manière de contourner les difficultés liées au jeu d'instruction x86. L'idée générale est de ne pas chercher à présenter l'interface d'un véritable processeur

x86 au sein de la machine virtuelle, mais une interface légèrement modifiée qui sera plus simple à émuler. En pratique cela revient à définir une interface entre le système d'exploitation s'exécutant dans la machine virtuelle et l'hyperviseur. Par analogie aux appels système, nommés *syscall*, ces appels à l'hyperviseur sont nommés *hypercall.*

Une fois de plus, c'est une technique qui était déjà utilisée dans les *mainframes* dès les années soixante-dix. L'hyperviseur IBM VM/370 (descendant de CP/CMS évoqué précédemment) ne virtualisait pas l'instruction DIAGNOSE qui déclenchait normalement des opérations de diagnostique. À la place, toute utilisation de cette instruction dans une machine virtuelle était interprétée comme un hypercall. L'objectif était d'améliorer les performances des systèmes d'exploitation virtualisés en remplaçant une séquence d'instruction causant de nombreuses interruptions logicielles par un unique hypercall.

Plus récemment, ce principe a aussi été appliqué à l'architecture x86. Certains hyperviseurs ne virtualisent pas l'ensemble du jeu d'instruction et proposent des hypercalls à la place de certaines instructions sensibles. Cela permet à la fois de résoudre le problème des instructions x86 non virtualisables et de minimiser le coût de virtualisation en évitant les interruptions logicielles.

L'inconvénient est que les machines virtuelles ne peuvent accueillir que des programmes qui ont été modifiés pour utiliser l'interface proposée par l'hyperviseur. Dans certains cas de paravirtualisation poussée, cette restriction concerne même les applications [38]. D'autres hyperviseurs, tels que Xen [39], ne nécessitent que des modifications assez légères qui sont cantonnées au système d'exploitation. Le portage initial de Linux sur cet hyperviseur a ainsi nécessité un patch de quelques milliers de lignes de code seulement. Le problème est que chaque hyperviseur a sa propre interface, et que ces interfaces ne sont pas aussi stables que les véritables interfaces matérielles. Ces modifications sont donc difficiles à maintenir, et les développeurs de Linux ont été initialement très réticents à ce qu'elles soient intégrées dans les noyaux standards.

Une couche d'abstraction a finalement été développée pour faciliter cette tâche : les *paravirt-ops*. Chacune de ces opérations est implémentée pour le jeu d'instruction x86 natif ainsi que pour les différents hyperviseurs supportés. Lorsque cette couche d'abstraction est activée, Linux détermine au démarrage s'il s'exécute au-dessus d'un hyperviseur paravirtualisé et fait pointer les paravirt-ops sur l'implémentation correspondante. Néanmoins, cette approche reflète la difficulté de mise en oeuvre de la paravirtualisation puisque plus de 70 paravirt-ops ont dû être introduites. Des outils d'aide à la paravirtualisation ont d'ailleurs été proposés[40].

Enfin, comme dans le cas de la translation binaire, le perte de la protection mémoire par segmentation lors du passage au 64 bits contraint l'hyperviseur à devoir utiliser la pagination pour se protéger des machines virtuelles. De ce fait, l'espace noyau et l'espace utilisateur des machines virtuelles doivent se partager le mode d'exécution non privilégié et ne peuvent être projetés en mémoire en même temps. A chaque appel système, un changement de contexte est nécessaire pour passer de l'un à l'autre ce qui cause un surcoût important.

2.2.1.4 Virtualisation assistée par le matériel

Suite à l'intérêt croissant pour la virtualisation depuis les années 2000, des instructions supplémentaires ont été introduites par Intel (VT) puis par AMD (SVM/AMD-V) afin de permettre aux processeurs x86 d'être virtualisés par la technique du *trap and emulate*. Ces deux extensions fonctionnent de manière similaire.

Elles permettent au processeur de travailler dans deux modes d'exécution supplémentaires que nous appellerons hôte et invité qui sont destinés à accueillir respectivement l'hyperviseur et les machines virtuelles. Au sein de chacun de ces deux modes d'exécution, l'ensemble des fonctionnalités du jeu d'instruction x86, telles que les 4 niveaux de privilèges, sont disponibles. En mode hôte, le processeur se comporte de manière identique au mode standard mais dispose d'une capacité supplémentaire : il peut déclencher un changement de contexte et passer en mode invité. Dans ce second mode, certaines instructions sensibles déclenchent des interruptions logicielles qui se traduisent par un retour en mode hôte. Ces transitions entre hôte et invité sont régies par une structure de contrôle stockée en mémoire et accessible par l'hôte.

Cette structure contient :

– **L'état du processeur en mode hôte et en mode invité**. La structure de contrôle permet de stocker les registres sensibles du processeur pour chacun des deux modes. Lors d'une transition d'un mode à l'autre, les valeurs de ces registres pour le mode courant y sont automatiquement sauvegardées, et les valeurs correspondant au nouveau mode d'exécution sont restaurées.

– **La cause de la dernière sortie du mode invité**. Cette donnée simplifie le travail de l'hyperviseur qui doit émuler le résultat de l'instruction fautive le plus efficacement possible. Elle lui évite d'avoir à analyser l'état complet du processeur virtuel pour déterminer l'action à entreprendre.

– **Des options de configuration des transitions**. Ces options permettent notamment à l'hyperviseur de contrôler, dans une certaine mesure, quelles instructions sensibles déclenchent des trappes et quels registres sensibles sont automatiquement sauvegardés et restaurés.

Ces extensions rendent donc les processeurs x86 conformes à l'ensemble des critères de Goldberg, la déprivilégisation se faisant en exécutant les machines virtuelles en mode invité, plutôt qu'en utilisant les niveaux de privilèges inférieurs du jeu d'instruction x86 d'origine. Elles permettent de mettre en place une virtualisation de type *trap and emulate* qui est bien plus simple à implémenter que la translation binaire et limite donc le risque de bugs et failles de sécurités. C'est par ailleurs à l'heure actuelle la seule manière de virtualiser complètement et efficacement le mode 64 bits.

L'inconvénient principal de ce système est son manque de flexibilité, car il n'autorise pas certaines optimisations rendues possibles par la translation binaire telles que l'élimination des interruptions logicielles par adaptation dynamique de code. Cela est d'autant plus pénalisant que les changements de contexte entre hôte et invité sont assez lourds, sachant que la table des pages ainsi que de nombreux registres doivent être restaurés. De plus, l'hyperviseur doit déterminer de lui-même l'action d'émulation à entreprendre alors que la translation binaire permet d'insérer un appel spécifique à la couche d'émulation.

Cependant, puisque la plupart des registres sensibles sont répliqués dans le mode invité, le nombre d'instructions causant des interruptions logicielles est réduit. Par exemple l'instruction permettant de désactiver les interruptions n'est pas interceptée par l'hôte. L'information est simplement stockée dans l'état du processeur virtuel. Les interruptions destinées au processeur hôte seront elles toujours délivrées après avoir causé un retour en contexte hôte. De même les appels système ne demandent pas d'intervention de l'hyperviseur puisque les différents niveaux de privilèges sont disponibles au sein du mode invité.

En outre, le support matériel de la virtualisation ne cesse de s'affiner. Le coût des changements de contexte diminue à chaque version de processeur et de nouvelles fonctionnalités sont ajoutées. Citons notamment le support des identifiants d'espace d'adressage qui permettent

d'éviter de vider la TLB à chaque changement de contexte, ainsi que le support des tables des pages imbriquées (voir section 2.2.2.5).

Pour finir, il est possible de combiner la virtualisation assistée par le matériel avec une paravirtualisation des certaines opérations coûteuses, comme nous le verrons dans la section 2.2.2.3.

Ainsi, ces extensions du jeu d'instruction x86 permettent de le virtualiser efficacement et fidèlement tout en gardant un hyperviseur relativement simple. Elles sont supportées par la plupart des processeurs haut de gamme d'Intel et AMD qui sont utilisés dans les grappes de calcul actuelles.

2.2.2 Virtualisation de la mémoire

Les instructions x86 font référence à la mémoire avec des adresses virtuelles qui sont traduites en adresses physiques par l'unité de gestion de mémoire. Celle-ci doit être virtualisée par l'hyperviseur pour que l'isolation des machines virtuelles soit assurée. Une formalisation du procédé à mettre en oeuvre a été proposée par Goldberg [34].

Soit R, l'ensemble des adresses physiques des ressources d'une machine, et V l'ensemble des adresses virtuelles utilisables par le processeur, Goldberg définit la fonction

$$\phi : V \longrightarrow R \cup \{i\}$$

qui associe à toute adresse $x \in V$ la ressource $y \in R$ correspondante ou une interruption logicielle i si aucune ressource n'est projetée à cette adresse. Cette fonction correspond donc à la traduction effectuée par l'unité de gestion mémoire. Dans le cas d'une machine virtuelle, les adresses résultantes sont les adresses de ressources physiques virtuelles exposées par l'hyperviseur. Ce dernier doit définir la correspondance entre ressources physiques virtuelles et ressources physique réelles. La méthode utilisée pour mettre en place cette correspondance caractérise les deux types d'hyperviseurs identifiés par Goldberg.

Hyperviseur de type I

Un hyperviseur de type I s'exécute en mode privilégié et est capable d'associer directement les ressources physiques virtuelles aux ressource physiques de l'hôte. Soit R les adresses physiques virtuelles, et R' les adresses physiques de l'hôte, cette association peut être définie par une fonction

$$f_I : R \longrightarrow R' \cup \{e\}$$

e étant une condition d'erreur survenant en cas d'utilisation d'une adresse physique virtuelle invalide.

La relation entre les adresses virtuelles utilisées par le processeur virtuel et les ressources physiques de la machine hôte est donc définie par

$$f_I \circ \phi : V \longrightarrow R' \cup \{i\} \cup \{e\}$$

Hyperviseur de type II

Un hyperviseur de type II s'exécute au-dessus du système d'exploitation de la machine hôte. Il ne peut généralement pas manipuler directement l'unité de gestion mémoire hôte et travaille uniquement avec des adresse virtuelles. Il associe donc les adresses physiques de la machine virtuelle à des adresses virtuelles de l'hôte. Soit R les adresses physiques virtuelles, et V' les adresses adresses virtuelles de l'hôte, cette association peut être définie par une fonction

$$f_{II} : R \longrightarrow V' \cup \{e\}$$

La relation entre les adresses virtuelles utilisées par le processeur virtuel et les ressources physiques de la machine hôte est donc définie par

$$\phi' \circ f_{II} \circ \phi : V \longrightarrow R' \cup \{i\} \cup \{e\}$$

où ϕ'correspond à traduction effectuée par l'unité de gestion mémoire hôte, contrôlée par le système d'exploitation.

Cette formalisation s'applique parfaitement aux différents procédés de virtualisation de la mémoire mis en oeuvre aujourd'hui pour l'architecture x86. Nous les présentons dans cette section après avoir rappelé le fonctionnement de l'unité de gestion mémoire x86

2.2.2.1 L'unité de gestion mémoire x86

En mode protégé, les processeurs x86 peuvent contrôler l'adressage mémoire par l'intermédiaire de deux mécanismes complémentaires : la segmentation et la pagination. Les adresses virtuelles manipulées par le processeur, appelées adresses logiques, sont traduites en adresses linéaires par la segmentation, et ces adresses linéaires sont ensuite traduites en adresses physiques par la pagination.

Segmentation

Comme son nom l'indique, la segmentation partitionne l'espace des adresses logiques en un ensemble de segments. Chaque segment est défini par sa base, c'est-à-dire son adresse linéaire de départ, et sa taille. Le calcul des adresses linéaires à partir des adresses logiques se fait donc simplement par un décalage égal à l'adresse de base du segment utilisé.

Différents segments sont mis place par le système d'exploitation et le choix du segment utilisé se fait automatiquement en fonction du type de référence mémoire qui est effectué. Par exemple, les adresses logiques faisant référence à des instructions utilisent le segment de code (CS pour *Code Segment*) tandis que celles faisant référence aux données utilisent le segment de données (DS pour *Data Segment*). Il est aussi possible de spécifier manuellement le segment utilisé par chaque adresse logique.

En outre, les segments offrent des fonctionnalités de protection mémoire. Les adresses logiques utilisables sont limitées par la taille des segments et le niveau de privilège courant doit être supérieur au niveau de privilège minimum défini par le segment. Il est par ailleurs possible de contrôler les permissions en lecture, écriture et exécution de chaque segment.

Cependant la segmentation est très peu utilisée par les systèmes d'exploitation actuels qui se contentent généralement de segments « plats » assurant l'égalité entre adresses logiques et adresses linéaires. Cela explique que le support de la segmentation ait été réduit au minimum en mode 64 bits : les segments n'ont plus de taille et la plupart des segments ne peuvent avoir que zéro comme adresse de base. De ce fait, par souci de concision, nous ne nous intéressons par la suite qu'à la pagination.

Pagination

La pagination subdivise quant à elle l'espace des adresses linéaires en pages, c'est-à-dire en intervalles d'adresses de taille fixe, typiquement 4KB [5]. L'unité de gestion mémoire traduit alors les adresses linéaires en adresses physiques grâce à la table des pages qui associe chaque page à une zone de taille équivalente en mémoire physique. La table des pages est stockée en mémoire de manière hiérarchique et l'adresse physique du premier niveau de la table doit être indiquée dans le registre CR3 du processeur.

Le parcours des différents niveaux de la table des pages étant coûteux, le TLB permet de garder en cache les dernières correspondances entre adresses logiques et physiques utilisées. Lors d'un changement de table des pages, c'est-à-dire lorsque la valeur de CR3 change, ce cache doit être intégralement vidé [6] ce qui implique une importante pénalité lors des accès mémoire suivants.

La pagination permet aussi d'assurer la protection mémoire. Chaque page peut être marquée comme privilégiée, ce qui la rend inaccessible par les codes s'exécutant avec un CPL égal à 3, et les accès en lecture, écriture ou exécution peuvent être restreints. Tout accès à une page non autorisée, ou ne disposant pas de projection en mémoire physique valide déclenche une interruption logicielle.

2.2.2.2 Virtualisation purement logicielle

L'unité de gestion mémoire peut être virtualisée de manière purement logicielle dans le cadre de la translation binaire. Pour cela, la mémoire physique attribuée à la machine virtuelle est projetée dans le même espace d'adressage que les blocs de code traduits, généralement selon un intervalle contigu. La virtualisation de l'unité de gestion de mémoire affecte le traitement de chaque instruction référençant la mémoire. Leur traduction doit être précédée par une suite d'instructions permettant de calculer l'adresse virtuelle de l'hôte correspondant à l'adresse virtuelle de l'invité référencée. Chaque accès mémoire conduit donc à un parcours logiciel de la table des pages de l'invité afin d'obtenir l'adresse physique virtuelle correspondante. L'adresse virtuelle à utiliser dans l'hôte s'obtient ensuite en y ajoutant l'adresse à laquelle la mémoire de la machine virtuelle est projetée. Le coût du parcours de la table des pages de l'invité peut néanmoins être amorti par l'utilisation d'un cache stockant les dernières translations mémoire utilisées, équivalent à un TLB logiciel.

Selon la formalisation de Goldberg, cela revient à calculer logiciellement le résultat de $f_{II} \circ \phi$ pour chaque accès mémoire. Seule ϕ' est appliquée matériellement par l'unité de gestion de gestion mémoire. Cette technique est donc peu efficace et n'est généralement utilisée que pour émuler une architecture différente de l'architecture hôte.

5. Des grosses pages, de 4MB (en mode 32 bits) ou 2MB (en mode 64 bits) sont aussi supportées.

6. Ce comportement peut être évité pour les pages dont la traduction reste la même dans toutes les tables de pages en les identifiant comme pages globales.

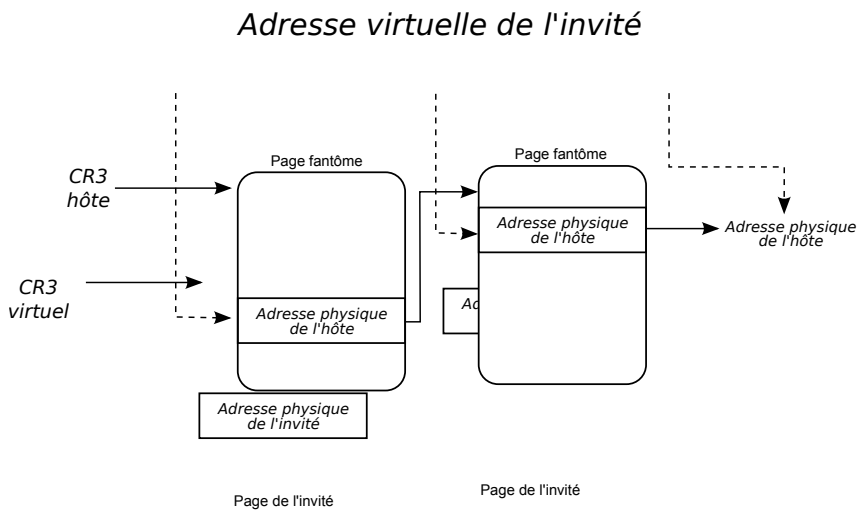


FIGURE 2.3: Table des pages fantôme

2.2.2.3 Table des pages fantôme

Lorsque le code de la machine virtuelle s'exécute directement sur le processeur hôte, l'hyperviseur ne peut laisser le système d'exploitation invité installer sa propre table des pages dans l'unité de gestion mémoire. D'une part parce que cela empêcherait d'isoler correctement la machine virtuelle, et d'autre part parce que les tables des pages construites par les systèmes d'exploitation virtualisés font référence à des adresses physiques virtuelles.

Une solution consiste à découpler la table des pages manipulée par le système d'exploitation invité de la table des pages qui est réellement installée dans l'unité de gestion mémoire. Cette dernière est alors appelée table des pages fantôme. Elle est construite en convertissant les adresses physiques virtuelles indiquées dans la table des pages de l'invité en adresses physiques réelles. Si nous reprenons le formalisme de Goldberg introduit précédemment, la table des pages fantôme contient donc la composition $f \circ \phi$, calculée de manière logicielle par l'hyperviseur. Pendant l'exécution de la machine virtuelle, l'unité de gestion mémoire de l'hôte pointe sur cette table des pages fantôme ce qui permet d'appliquer matériellement $f \circ \phi$ à toutes les adresses virtuelles utilisées (voir Figure 2.3).

La difficulté réside dans le maintien efficace de la synchronisation entre la table des pages fantôme et la table des pages utilisée par l'invité. La technique de base consiste à intercepter les modifications du registre CR3 du processeur virtuel afin d'analyser la nouvelle table des pages et de mettre en place la table des pages fantôme correspondante. Cela ne pose pas de problème dans le cadre des différentes techniques de virtualisation du processeur évoquées précédemment car c'est une opération privilégiée qui déclenche aussi une sortie du mode invité si on utilise le support matériel de la virtualisation.

Il faut en outre pouvoir détecter si une entrée de la table des pages courante est modifiée. Pour cela, les pages de la table des pages de l'invité sont placées en lecture seule dès qu'elles ont été analysées par l'hyperviseur, et que les pages fantôme correspondantes ont été construites. De cette manière, toute modification ultérieure de la table des pages par l'invité déclenchera une interruption logicielle qui permettra à l'hyperviseur de mettre a jour l'entrée correspondante dans la table des pages fantôme.

Cette opération est plus compliquée qu'il n'y paraît car ces pages de table des pages peuvent être rendues accessibles par n'importe quelle projection mise en place par l'invité. Il faut donc

passer en lecture seule toutes les projections correspondantes dans les différentes tables des pages fantôme mises en jeu.

Tous ces mécanismes se révèlent assez coûteux ce qui a conduit au développement de diverses optimisations :

Cache de pages de table des pages fantôme

Typiquement un système d'exploitation utilise une table des pages par processus. La table des pages courante est donc échangée à chaque commutation de processus. Comme il est trop coûteux de reconstruire toute la table des pages fantôme à chaque fois, on utilise un cache de pages de table des pages fantôme (ces dernières seront appelées par la suite pages fantômes). Ce cache permet de retrouver rapidement la page fantôme correspondant à une page de table des pages de l'invité.

Néanmoins il faut être en mesure de déterminer quand une page de l'invité n'est plus utilisée comme page de table des pages afin de pouvoir restaurer l'accès en écriture et supprimer la page fantôme correspondante du cache. Malheureusement, seules des heuristiques peuvent être utilisées à cet effet. On peut par exemple se baser sur le type d'accès effectué par l'invité : s'il accède à une page depuis un contexte non privilégié, ou de façon non alignée sur un mot, cela laisse penser qu'elle n'est probablement plus utilisée pour une table des pages.

Mise à jour paresseuse

Le coût des interruptions logicielles peut devenir prohibitif si le système d'exploitation invité effectue un grand nombre de modifications successives de la table des pages. C'est typiquement ce qui se passe lors d'une séquence de type fork/exec. Pour pallier ce problème un algorithme de mise à jour paresseuse peut être utilisé.

Le principe est de restaurer les droits en écriture sur les pages de la table des pages de l'invité qui déclenchent un trop grand nombre d'interruptions logicielles. Le système d'exploitation est alors libre de mettre à jour sa table des pages mais les modifications ne sont plus répercutées sur l'unité de gestion mémoire de l'hôte. Cela ne pose pas de problème car on peut exploiter une contrainte que tous les systèmes d'exploitation doivent respecter du fait de l'existence d'un TLB : lors d'une modification d'une entrée de table des pages le TLB doit être purgé des entrées périmées avant que les nouvelles projections puissent être utilisées. Ainsi, deux événements vont déclencher une resynchronisation des pages fantômes :

1. L'utilisation d'instructions causant une invalidation partielle ou totale du TLB (INVLPG ou MOV CR3).
2. Un défaut de page correspondant à une entrée nouvellement crée par l'invité, mais pas encore répercutée sur la page fantôme correspondante. En effet la mise en place d'une projection pour une adresse précédemment invalide ne nécessite pas de vider le TLB. Aussi, l'hyperviseur vérifie si l'on se trouve dans ce cas de figure à chaque défaut de page survenant dans l'invité. Le cas échéant, l'entrée correspondante est ajoutée à la page fantôme et le défaut de page n'est pas répercuté chez l'invité.

En dépit de ces différentes optimisations, le coût du maintien des tables des pages fantômes reste important lorsque les projections mémoire mises en place par le système d'exploitation

invité changent fréquemment. Ce surcoût est amplifié pour les machines virtuelles multiprocesseurs, puisque les tables des pages des différents processeurs sont indépendantes. Il faut donc maintenir simultanément autant de tables des pages fantômes que de processeurs virtuels ce qui est coûteux à cause des synchronisations nécessaires. Cela est d'autant plus vrai dans le cas d'un programme multi-threadé puisque les projections mémoire doivent être maintenues cohérentes entre les processeurs exécutant les différents threads ce qui cause de nombreuses invalidations de TLB simultanées.

2.2.2.4 Paravirtualisation

On peut se passer de l'utilisation d'une table des pages fantôme si on utilise un système d'exploitation paravirtualisé. En effet, un tel système peut être modifié de manière à n'accéder qu'aux pages physiques qui lui ont été réservée. Dans ce cas, il n'est pas nécessaire de virtualiser la table des pages et on peut se contenter de restreindre l'accès en écriture. Le système d'exploitation paravirtualisé effectuera les modifications de la table des pages par des hypercalls.

Ce mécanisme permet de limiter les différents surcoûts rencontrés lors de l'utilisation d'une table des pages fantôme. L'interface des hypercalls peut être optimisée afin de permettre un traitement par lot des modifications de table des pages et limiter ainsi le nombre de changements de contexte. En outre, le système d'exploitation invité indique directement les entrées à placer dans la table des pages car il connait les adresses des pages physiques qui lui sont attribuées. Le travail de l'hyperviseur est donc fortement simplifié puisqu'il doit simplement vérifier si ces entrées sont valides. Il doit principalement s'assurer que les adresses physiques utilisées correspondent bien à des pages de la machine virtuelle, et que l'invité ne tente pas d'établir une projection permettant d'écrire dans une page devant rester en lecture seule, telles que les pages de la table des pages.

En revanche, cette technique limite quelque peu la flexibilité offerte par la virtualisation puisque l'hyperviseur ne peut plus changer les pages physiques qui sont attribuées à la machine virtuelle comme il le souhaite. Cela empêche par exemple de déporter la mémoire des machines virtuelle sur le disque dur, si l'on souhaite sur-allouer la mémoire de l'hôte [7], ou de migrer les machines virtuelles d'un hôte à l'autre de manière transparente.

2.2.2.5 Support matériel des table des pages imbriquées

Nous avons vu que certains processeurs x86 récents intègrent un support matériel de la virtualisation à travers des extensions du jeu d'instruction qui ne cessent de gagner en fonctionnalités. Les dernières générations de processeur [8] permettent à l'unité de gestion mémoire de traduire les adresses virtuelles de l'invité en adresses physiques, et ce sans que l'hyperviseur ait à intervenir lors des modifications de la table des pages de l'invité. Pour cela, l'unité de gestion mémoire a été modifiée afin de prendre en compte deux tables des pages imbriquées.

La première table des pages est mise en place par l'hyperviseur et indique la correspondance entre adresses physiques virtuelles et adresses physiques réelles, c'est-à-dire la fonction f de Goldberg. Cette correspondance est indépendante de toute opération effectuée par la machine

7. Allouer plus de mémoire pour l'ensemble des machines virtuelles qu'il y a de mémoire disponible sur l'hôte

8. Cette fonctionnalité est disponible chez AMD sous le nom Nested Page Table depuis la microarchitecure Barcelona, sortie en 2007, et chez Intel sous le nom Extended Page Table à partir de la microarchitecure Nehalem sortie en 2008

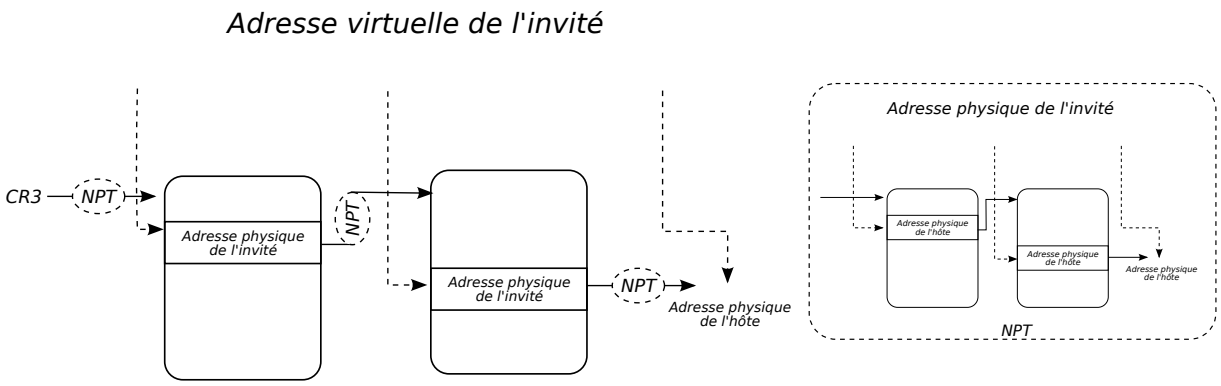


FIGURE 2.4: Support matériel des tables des pages imbriquées

virtuelle. Elle peut donc être fixée une fois pour toute, ou au contraire évoluer dynamiquement en fonction de la quantité de ressources que l'hyperviseur souhaite allouer à chaque machine virtuelle.

La seconde table des pages est entièrement contrôlée par l'invité. Avec cette extension, les modifications des registres sensibles liés à l'unité de gestion mémoire, tel que le registre CR3, ne déclenchent plus d'interruptions logicielles lorsque le processeur s'exécute en mode invité. Le système d'exploitation invité peut donc installer sa propre table des pages, indiquant la correspondance entre adresses virtuelles et adresses physiques virtuelles, soit la fonction ϕ.

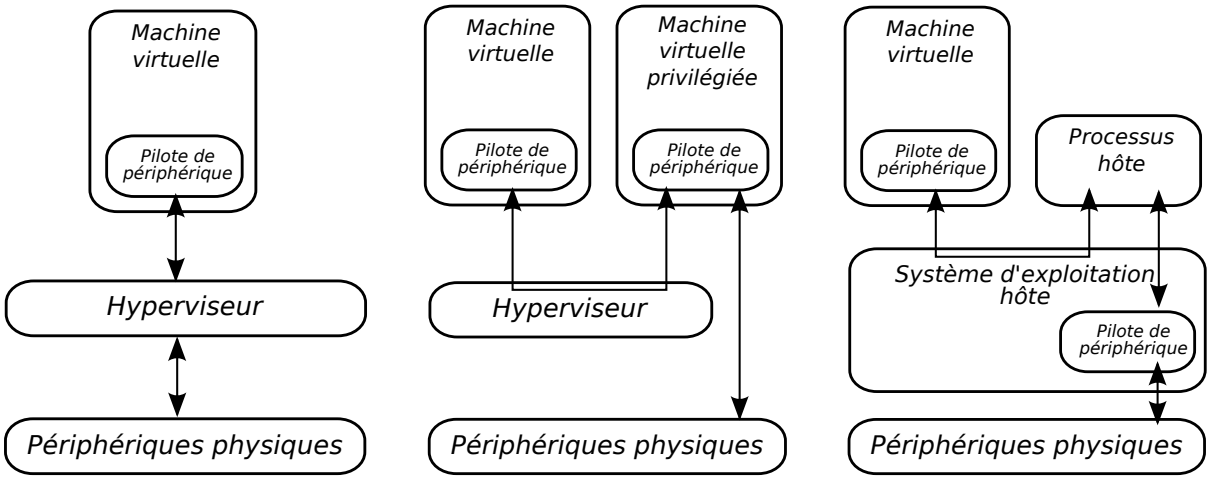
Lors des accès mémoire, l'unité de gestion mémoire va parcourir ces deux tables des pages pour effectuer la translation d'adresse (voir Figure 2.4). Cette solution revient donc à supporter matériellement et dynamiquement le calcul de la composition $f \circ \phi$.

Notons que ce calcul est assez coûteux puisque, en mode 64 bits, chaque table des pages est composée de 4 niveaux de hiérarchie. Or, dans la table des pages de l'invité, chaque niveau de hiérarchie fait référence au niveau suivant par des adresses physiques virtuelles. Pour trouver l'adresse physique réelle correspondante, la table des pages de l'hôte doit être parcourue. Cela signifie que, à chaque translation d'adresse, l'unité de gestion mémoire doit parcourir 5 fois les 4 niveaux de table des pages de l'hôte : une fois pour chaque niveau de la table des pages de l'invité et une dernière fois pour le résultat de la translation initialement calculée. Chaque translation d'adresse demande donc 24 accès mémoire, contre seulement 4 en temps normal.

L'efficacité de cette solution dépend donc fortement de la capacité du TLB à minimiser le nombre de parcours de tables des pages. Pour cela, les TLBs ont été étendus afin de pouvoir garder en cache, d'une part, le résultat des translations d'adresses virtuelles de l'invité, et d'autre part, le résultat des translations d'adresses physiques de l'invité utilisées lors des parcours de table des pages. Par ailleurs, un cache de table des pages peut être introduit afin de minimiser le coût des accès mémoire lors des défauts de TLB [41]. Enfin, l'utilisation de grosses pages peut être intéressante pour minimiser le nombre d'entrées consommées dans le TLB. Malheureusement, la plupart des systèmes d'exploitation n'ont pas encore un support très mûr des grosses pages, et il est difficile de les utiliser efficacement dans des applications.

Discussion

Le choix entre l'utilisation de tables de pages imbriquées ou fantômes relève donc d'un compromis. Les tables des pages fantômes entraînent un surcoût important à chaque fois que l'invité modifie ses projections mémoire et de la mémoire est gaspillée par le cache de pages fantômes.



(a) Hyperviseur de type I

(b) Hyperviseur de type I avec *driver domain*

(c) Hyperviseur de type II

FIGURE 2.5: Architectures d'hyperviseur pour l'émulation de périphériques

Elles sont en outre difficiles à mettre en oeuvre et complexifient fortement l'hyperviseur. En revanche, contrairement aux tables des pages imbriquées, elles n'entraînent aucune perte de performances si la table des pages de l'invité reste constante.

La meilleure technique à mettre en oeuvre dépend donc de la charge de travail que les machines virtuelles ont à traiter. Une machine virtuelle constituée d'une unique application ayant un schéma d'allocation relativement statique pourra être plus efficace avec une table des pages fantôme. A l'inverse les tables des pages imbriquées seront nettement plus performantes si la machine virtuelle exécute de nombreux processus dont les allocations mémoire sont très dynamiques.

2.2.3 Virtualisation des périphériques

Le processeur et la mémoire constituent le coeur d'une machine mais ils ne peuvent être exploités sans périphérique d'entrée/sortie. Nous avons par exemple déjà souligné l'importance des périphériques réseaux haute performance dans le cadre des grappes de calcul. Il est donc primordial que les hyperviseurs puissent exposer des périphériques performants dans les machines virtuelles.

Nous étudions donc dans cette section les procédés mis en oeuvre à cet effet. Nous présentons tout d'abord les différentes techniques s'apparentant à de l'émulation puis nous nous intéressons à celles qui permettent à une machine virtuelle de contrôler directement un périphérique existant sans causer de problème de sécurité.

2.2.3.1 Émulation

Nous qualifions d'émulation les méthodes de virtualisation de périphérique avec lesquelles l'hôte reste le seul à pouvoir contrôler les périphériques physiques. Les commandes envoyées au périphérique virtuel sont toutes traitées par l'hyperviseur qui se charge d'agir sur les périphériques physiques en conséquence. En particulier cette technique implique que l'hyperviseur soit capable de piloter les périphériques physiques en question, de manière directe ou

indirecte. Pour cela, différentes stratégies peuvent être adoptées en fonction de l'architecture de l'hyperviseur.

Hyperviseur de type I

Les hyperviseurs de type I s'exécutent seuls sur la machine hôte. Une première solution consiste à implémenter le support des périphériques directement dans l'hyperviseur (voir Figure 2.5a). Cela permet notamment d'éviter les problèmes de performances liés à une interface inadaptée et de laisser l'hyperviseur en charge de l'ensemble des accès privilégiés. En revanche, la liste des périphériques supportés par un hyperviseur est nécessairement beaucoup plus courte que celle des périphériques pris en charge par un système d'exploitation généraliste. Ce problème peut être contourné en utilisant une machine virtuelle privilégiée, parfois appelée *driver domain*, ayant un accès direct aux périphériques (selon le procédé décrit section 2.2.3.2). L'hyperviseur peut alors déléguer la gestion des périphériques au système d'exploitation exécuté dans la machine virtuelle et lui soumettre les différentes opérations nécessaires à l'émulation des périphériques virtuels (voir Figure 2.5b).

Hyperviseur de type II

Les hyperviseurs de type II s'exécutent au sein d'un système d'exploitation existant. Ils peuvent donc réutiliser les différentes abstractions fournies par le système d'exploitation hôte pour interagir avec tous les périphériques supportés (voir Figure 2.5c). On assure ainsi une grande portabilité matérielle de l'hyperviseur sans demander de développement important. En contrepartie, les interfaces proposées par les systèmes d'exploitation ne sont pas toujours bien adaptées à l'implémentation d'un hyperviseur. Cela peut par exemple causer des changements de contexte, ou des copies mémoire inutiles, et donc dégrader les performances des périphériques virtuels. Aussi, du fait de l'importance croissante de la virtualisation, des interfaces nouvelles ont été proposées pour améliorer les performances des hyperviseurs de type II [42].

Un hyperviseur peut émuler un périphérique existant, ou définir un périphérique présentant un interface nouvelle, ce qui est une forme de paravirtualisation. Nous étudions maintenant ces deux cas de figures et présentons leurs avantages et inconvénients respectifs.

Émulation d'un périphérique existant

Les processeurs x86 peuvent interagir avec les périphériques selon deux procédés.

1. Les entrées/sorties par port (*port mapped I/O*) utilisent un espace d'adresses dédié, les adresses de port. Des instructions dédiées, IN et OUT, permettent respectivement de lire ou d'écrire une donnée à une adresse de port spécifique. Ces requêtes sont transmises sur un bus d'entrées/sorties qui permet à chaque périphérique de réagir aux requêtes correspondant aux ports qui lui sont attribués.
2. Les entrées/sorties projetées en mémoire (*memory mapped I/O*) permettent au processeur d'adresser indifféremment mémoire centrale et périphériques. Une partie de l'espace des adresses mémoire physiques est réservée à chaque périphérique. Lorsque le processeur lit ou écrit à ces adresses, en utilisant des instructions standard, les données sont relayées au périphérique correspondant et évitent le contrôleur mémoire.

Les instructions d'entrées/sorties par port étant privilégiées, elles peuvent facilement être interceptées par l'hyperviseur quelle que soit la technique de virtualisation du processeur employée. En ce qui concerne les entrées/sorties projetées en mémoire, l'hyperviseur peut empêcher l'invité de mettre en place des projections mémoire lisibles ou écrivables référençant les adresses physiques assignées aux périphériques virtuels. Les accès à cette zone mémoire déclenchent alors des interruptions logicielles qui permettent à l'hyperviseur d'émuler le comportement attendu.

Ces deux mécanismes suffisent à émuler n'importe quel périphérique existant. Prenons l'exemple des périphériques basés sur le bus standard PCI (pour P*eripheral Component Interconnect*). Un système d'exploitation recherche les périphériques PCI présents sur la machine par des lectures à des adresses d'entrée/sortie pré-déterminées. Ces lectures sont interceptées par l'hyperviseur qui peut alors retourner les identifiants de modèles de périphériques existants. Le système invité configure ensuite par le même mécanisme les adresses d'entrées/sorties, par port ou projetées en mémoire, utilisées par les différents périphériques détectés. Cela permet à l'hyperviseur de protéger les adresses mémoire correspondantes et de router les interruptions logicielles résultantes sur les bonnes fonctions d'émulation de périphérique.

Examinons plus en détail le cas de l'émulation d'un périphérique réseau tel qu'une carte Ethernet. Pour envoyer des données, le pilote de périphérique doit généralement construire une file de descripteurs de paquets en mémoire. Chaque descripteur de paquet contient notamment l'adresse en mémoire des données à envoyer ainsi leur taille. Des entrées/sortie sont utilisées, à l'initialisation, pour indiquer l'adresse de la file à la carte Ethernet, puis à chaque ajout de descripteurs dans la file, pour indiquer la nouvelle position du pointeur de queue. Le périphérique réseau utilise alors des accès DMA pour lire le contenu de la file, puis pour lire les données à envoyer à partir de l'adresse indiquée par le descripteur. Il met ensuite à jour le pointeur de tête de file, qui peut être lu par une entrée/sortie, et transmet les données sur le réseau. La réception de données fonctionne avec une file similaire, le périphérique réseau utilisant alors des DMAs pour écrire les paquets reçus aux adresses pointées par les descripteurs. Enfin, des interruptions peuvent être envoyées au processeur pour signaler la fin de l'envoi ou de la réception d'un paquet.

L'émulation d'un tel comportement est donc assez simple. Les opérations d'entrée/sortie étant interceptées par l'hyperviseur, celui-ci peut directement lire et écrire dans la mémoire de l'invité pour simuler les DMAs. Il peut alors donner l'illusion que le périphérique virtuel est directement relié au réseau physique :

1. Les paquets émis dans la machine virtuelle sont retransmis tels quels par le périphérique hôte.
2. Le périphérique hôte est configuré en mode promiscuité (*promiscuous mode*) afin qu'il prenne en compte les paquets reçus dont l'adresse Ethernet de destination n'est pas la sienne. Ces paquets peuvent alors être relayés à la bonne machine virtuelle.

Des interruptions virtuelles sont par ailleurs injectées si nécessaire pour signaler la fin du traitement d'un paquet.

L'avantage majeur de l'émulation est qu'elle permet de virtualiser n'importe quel système d'exploitation supportant le matériel émulé. Il est en outre relativement simple d'émuler une large gamme de périphériques différents que ce soit pour assurer une compatibilité maximum ou pour permettre la simulation d'une configuration de machine précise, à des fins de test.

Cette technique est en revanche coûteuse, notamment à cause des nombreux changements de contexte causés par les instructions d'entrée/sortie et les interruptions. Notons que ce coût peut être mitigé si le processeur est virtualisé par translation binaire grâce aux techniques d'adaptation de code.

Paravirtualisation

L'hyperviseur peut exposer des périphériques paravirtualisés fournissant les principales fonctionnalités nécessaires à une machine, notamment la connectivité réseau et le stockage disque. L'interface de ces périphériques, basée sur des hypercalls, est alors conçue dans l'optique de minimiser le nombre de changements de contexte déclenchés à chaque opération.

Comme dans le cas des processeurs, cette technique permet de gagner en performance au prix d'une perte de compatibilité avec les systèmes d'exploitation non modifiés. Cependant, c'est une forme de paravirtualisation beaucoup plus légère car les systèmes d'exploitation prévoient généralement des mécanismes pour charger dynamiquement des pilotes de périphériques supplémentaires. Distribuer des pilotes de périphériques virtuels pour les principaux systèmes d'exploitation permet donc d'assurer une assez bonne compatibilité. Ces pilotes sont en outre très simples car l'interface des périphériques paravirtualisés est généralement plus aisée à exploiter que celle d'un périphérique réel. Enfin, il est toujours possible de se rabattre sur l'émulation de périphériques standards lorsque l'invité ne reconnaît pas les périphériques paravirtualisés. Ainsi, la plupart des hyperviseurs proposent des périphériques paravirtualisés en option pour améliorer les performances des systèmes qui les supportent.

2.2.3.2 Accès direct

Les techniques de virtualisation de périphérique par accès direct permettent que la majeure partie des instructions d'entrée/sortie émises par la machine virtuelle soient directement reçues par le périphérique hôte correspondant.

Des méthodes proches de la déprivilégisation ont par exemple été proposées. L'idée est de n'intercepter que certaines adresses d'entrée/sortie correspondant à des opérations sensibles, et d'accorder à l'invité un accès direct aux autres adresses. Cependant, la sensibilité des opérations dépend de l'état courant du périphérique. Aussi, pour chaque modèle de périphérique virtualisé, il faut construire un automate à états permettant de déterminer les opérations à intercepter et les performances ne sont pas toujours au rendez-vous car elles causent de nombreux changements de contexte. En outre, si le périphérique est partagé entre plusieurs machines virtuelles, il faut sauvegarder et restaurer l'état complet du périphérique lors de chaque changement de contexte entre machines virtuelles ce qui est assez coûteux.

Cette technique est donc peu utilisée actuellement et on se contente généralement de donner un accès complet au périphérique tout en s'appuyant sur des extensions matérielles pour assurer l'isolation des machines virtuelles. Par ailleurs, certains périphériques supportent matériellement la virtualisation, ce qui permet de les assigner simultanément à plusieurs machines virtuelles. Nous examinons maintenant ces deux mécanismes.

Assignation d'un périphérique

L'assignation de périphérique ne correspond pas réellement à de la virtualisation. Elle permet de dédier un périphérique de l'hôte à une machine virtuelle qui peut alors l'utiliser à sa guise. Nous avons déjà évoqué l'intérêt de cette méthode dans le cadre des hyperviseurs de type I (voir section 2.2.3.1). Plus généralement elle peut être utilisée lorsque l'on souhaite tirer parti des performances natives du périphérique assigné. Il reste cependant un certain surcoût lié à la gestion des interruptions qui ne peuvent être délivrées directement à la machine virtuelle. Chaque interruption provoque un retour à l'hyperviseur, qui en analyse la source, et détermine le cas échéant le processeur virtuel dans lequel l'interruption doit être injectée.

Outre le fait de ne pouvoir partager le périphérique entre plusieurs machines virtuelles, cette technique pose le problème de l'isolation des machines virtuelles pour les périphériques capables d'effectuer des DMAs. Il faut en effet s'assurer que le périphérique ne peut accéder qu'à la mémoire de la machine virtuelle à laquelle il a été assigné. En outre les adresses DMA communiquées au périphérique par le pilote invité seront des adresses physiques virtuelles, qui doivent encore être traduites en adresses physiques de l'hôte avant d'être soumises au contrôleur mémoire.

Ces problèmes peuvent être résolus par l'utilisation d'une unité de gestion de la mémoire pour les entrées/sorties (IOMMU pour *Input/Output Memory Management Unit*). Ces IOMMU, qui sont apparues sur l'architecture x86 en même temps que les extensions pour la virtualisation citées précédemment, permettent d'appliquer une translation d'adresse aux opérations DMA émises par un périphérique. Ainsi, à la manière des processeurs, les périphériques travaillent sur des adresses virtuelles qui sont traduites en adresses physiques par une table des pages dédiée. L'hyperviseur peut donc encoder la translation entre adresses physiques de l'invité et de l'hôte dans cette table des pages. Cependant, les IOMMU ne permettent pas encore de générer des exceptions récupérables en cas de défaut de page ce qui implique que toute la mémoire de l'invité doit être punaisée en mémoire en permanence du point de vue de l'hôte. L'hyperviseur perd donc la flexibilité de faire évoluer dynamiquement les ressources physiques associées à chaque machine virtuelle. Notons que cette difficulté peut être contournée par l'utilisation d'une interface paravirtualisée pour les DMAs : avant chaque opération DMA, la machine virtuelle peut signaler à l'hyperviseur les tampons mémoire utilisés afin qu'il punaise les pages mémoire touchées et mette en place les translations correspondantes dans l'IOMMU le temps du transfert.

Support matériel de la virtualisation

De plus en plus de périphériques prennent matériellement en charge la virtualisation. C'est notamment le cas des périphériques PCI-Express supportant la norme SR-IOV [43] (pour *Single Root I/O Virtualization*) qui sont capables de se multiplexer d'eux-même en certain nombre de *fonctions virtuelles*. Ces *fonctions virtuelles* opèrent comme autant de périphériques PCI-Express indépendants qui peuvent être assignés à différentes machines virtuelles. Notons que cette fonctionnalité ne règle pas les problèmes liés aux opérations DMA effectuées par des périphériques assignés. Elle doit donc être utilisée de paire avec une IOMMU et les restrictions évoquées précédemment concernant l'attribution des ressources aux machines virtuelles s'appliquent toujours.

Par ailleurs, certains périphériques ayant des capacités de type OS-bypass (voir section 1.1.2.2) peuvent être virtualisés de manière similaire même s'ils ne supportent pas la norme SR-IOV. En effet, ces périphériques sont prévus pour être contrôlés directement depuis l'espace utilisateur par plusieurs processus, et ce de manière efficace et sécurisée. Ce mécanisme peut tout aussi bien fonctionner si les processus appartiennent à des machines virtuelles distinctes [44]. Il faut utiliser pour cela un pilote de périphérique paravirtualisé qui délègue à l'hôte l'établissement des projections mémoire permettant à un processus de piloter le périphérique. Ces opérations étant généralement hors du chemin critique, l'impact sur les performances est négligeable.

2.3 Virtualisation et calcul haute performance : atouts et défis

Nous avons vu que l'intérêt pour la virtualisation a été relancé par la nécessité de consolider des serveurs, dont la puissance de calcul dépasse souvent les besoins d'une unique tâche. Cet

engouement a conduit à la mise au point de nombreuses techniques qui permettent désormais de virtualiser efficacement les machines basées sur l'architecture x86, prédominante dans les grappes d'aujourd'hui. Pourtant, la virtualisation est encore peu utilisée dans le cadre du calcul haute performance. La consolidation n'a en effet que peu d'intérêt pour exécuter des applications parallèles consommant l'intégralité des ressources de plusieurs machines. Néanmoins, la virtualisation apporte de nombreux autres avantages dont pourraient bénéficier les grappes. Dans cette section, nous commençons par présenter ces atouts, puis nous discutons des défis à relever pour que la virtualisation puisse être mise en oeuvre plus simplement et efficacement dans les grappes.

2.3.1 De nombreux atouts

La virtualisation présente de nombreux attraits qui pourraient faciliter l'exploitation d'une grappe de calcul. Nous présentons ici quelques cas d'utilisation intéressants.

2.3.1.1 Allocation dynamique des ressources matérielles

La virtualisation introduit une couche d'abstraction entre matériel et logiciel qui permet de faire varier dynamiquement et de manière transparente les ressources physiques dédiées à une tâche. Cela offre de nouvelles possibilités pour les ordonnanceurs de travaux et permet de décharger les utilisateurs de certaines contraintes.

En effet, actuellement, la plupart des ordonnanceurs de travaux [45] demandent aux utilisateurs de spécifier la quantité de ressources dont ils ont besoin, par exemple le nombre de coeurs et la quantité de mémoire, ainsi que la durée pendant laquelle ils souhaitent les utiliser. Ces ressources sont alors réservées statiquement selon un algorithme de type premier arrivé premier servi. Le problème est que, même avec des optimisations telles que le remblayage (*backfilling*) [46], un tel algorithme ne permet pas toujours de pleinement occuper les ressources de la grappe. Ce gaspillage de ressources est accentué par la difficulté pour les utilisateurs d'évaluer précisément leurs besoins [47]. Si la durée de la tâche est sous-évaluée, celle-ci sera interrompue avant son terme ce qui peut conduire à la perte de tous les calculs effectués. De ce fait, les utilisateurs auront tendance à sur-évaluer leurs réservations. En outre, en cas de fluctuation de la consommation de ressources dans le temps, l'utilisateur doit réserver, pour toute la durée de la tâche, une quantité de ressource correspondant au besoin maximum anticipé.

De nombreux travaux proposent d'utiliser des machines virtuelles pour améliorer cette situation [48, 49, 50, 51]. Cela permet par exemple à l'ordonnanceur de préempter n'importe quelle tâche en effectuant des protection/reprise de machines virtuelles et d'augmenter ainsi les opportunités de remblayage. De plus, la consommation réelle des ressources par les machines virtuelles peut être surveillée afin d'adapter dynamiquement la réservation des ressources à l'utilisation qui en est réellement effectuée par chaque tâche. On se rapproche quelque peu du fonctionnement d'un ordonnanceur préemptif de processus dans un système d'exploitation. Enfin les machines virtuelles peuvent être migrées dynamiquement pour atteindre différents objectifs. Par exemple, lorsque la grappe n'est pas utilisée à pleine capacité, on peut regrouper les machines virtuelles sur un nombre minimal de noeuds et éteindre les noeuds non utilisés pour économiser de l'énergie [52]. À l'inverse, si la priorité est de minimiser le temps de calcul, on peut choisir de les étaler au maximum sur les différents noeuds afin de maximiser la bande passante mémoire et la quantité de cache disponible pour chaque tâche.

La virtualisation simplifie aussi le travail des administrateurs lors des opérations de maintenance. Cela permet de les effectuer à tout moment sans interruption de service, en migrant

simplement au préalable les machines virtuelles vers un autre noeud. De même l'impact des pannes peut être minimisé, de manière réactive, en restaurant automatiquement l'état des machines virtuelles à partir d'une protection, ou de manière proactive si un risque de panne est détecté [53, 54].

2.3.1.2 Encapsulation des adhérences logicielles

Nous avons vu que la virtualisation offre plus de flexibilité que le multiplexage des utilisateurs proposé par un système d'exploitation standard. Cette caractéristique est très intéressante dans le cadre de grappes de calcul partagées entre des milliers d'utilisateurs ayant des besoins hétérogènes.

Cohabitation de plusieurs systèmes d'exploitation

L'utilisation de machines virtuelles permet de changer de système d'exploitation sans avoir à redémarrer la machine hôte, voir de faire cohabiter simultanément plusieurs systèmes d'exploitation différents. Cela permet par exemple de faciliter la transition d'une version à l'autre d'un système en proposant aux utilisateurs un choix entre les deux versions pendant la durée de la transition.

Dans le cadre du calcul haute performance, plusieurs hyperviseurs ont ainsi été développés spécifiquement pour faire cohabiter des systèmes d'exploitation minimaux dédiés aux applications exigeantes en performance avec des systèmes d'exploitation généralistes proposant plus de fonctionnalités [55, 56]. En poussant cette idée plus loin on peut envisager l'utilisation de systèmes d'exploitation conçus spécifiquement pour exécuter une classe d'application donnée et ne fonctionnant que sur un hyperviseur [57, 58]. Des systèmes d'exploitation ont par exemple été proposés pour exécuter efficacement des machines virtuelles Java au-dessus d'un hyperviseur[59, 60].

Déploiement et pérennisation d'application

Comme nous l'avons souligné dans la section 1.3.3, il est difficile de déployer rapidement une application sur grappe de calcul du fait des diverses dépendances mises en jeu. La virtualisation permet de simplifier cette étape, puisque chaque utilisateur peut installer l'environnement d'exécution nécessaire à ses applications au sein d'une machine virtuelle qu'il administre. On peut ainsi utiliser une image de machine virtuelle pour empaqueter une application avec toutes ses dépendances, que ce soit les bibliothèques, le système d'exploitation ou les divers fichiers de configuration. Cette machine virtuelle peut alors être exécutée à l'identique sur n'importe quelle machine hôte, ce qui permet d'une part d'utiliser facilement des services fournissant de la puissance de calcul à la demande, et d'autre part de pérenniser une application puisqu'on est ainsi assuré qu'elle continuera de fonctionner indépendamment de toute évolution de la pile logicielle hôte.

2.3.1.3 Simulation d'environnements et introspection

Dans de nombreux cas, il peut être utile d'exécuter une application dans un environnement logiciel ou matériel précis à des fins d'évaluation, de débuggage ou de portabilité. La virtualisation peut être utilisée pour simuler ces environnements.

Simulation d'environnement logiciel

L'utilisation de machines virtuelles permet de reproduire des environnements logiciels divers sur une même machine, de façon simple et sécurisée. Cela peut par exemple être mis à profit pour préparer le déploiement d'une application dans les mêmes conditions que sur la machine ciblée. De même, il peut être utile de simuler l'environnement logiciel utilisé par un utilisateur afin de reproduire un bogue. Pour les bogues les plus difficiles à reproduire, il est même possible d'enregistrer une trace de l'exécution d'une application ce qui permet ensuite de la rejouer de manière déterministe [61, 62]. Enfin la virtualisation simplifie la résolution de problèmes liés au code noyau [63] puisqu'il est possible de le déboguer à partir de l'hôte ou d'une autre machine virtuelle privilégiée.

Simulation d'environnement matériel

La virtualisation peut aussi être utilisée pour évaluer le comportement de logiciels lorsqu'ils sont soumis à des contraintes matérielles différentes. Par exemple, il peut être intéressant de tester le fonctionnement d'une application sur une machine disposant de moins de mémoire, notamment lors du développement d'applications de type out-of-core. De même, la virtualisation peut être mise à profit pour tester le passage à l'échelle d'applications distribuées sur un grand nombre de machines. Cela peut par exemple servir à mettre au point les logiciels servant à administrer la génération de grappe suivante alors que l'on ne dispose pas encore de grappe de cette envergure [64].

Enfin, il est possible de simuler une architecture matérielle plus simple que celle de l'hôte à des fins de portabilité ou d'efficacité. On peut ainsi considérer que l'hyperviseur Disco, évoqué section 2.1.2.2, permet de simuler une machine à accès mémoire uniforme à partir d'une machine NUMA. D'autres hyperviseurs permettent d'émuler une machine à mémoire partagée à partir d'une grappe en mettant en place une DSM [65, 66, 67]. On peut alors piloter toute la grappe à partir d'un unique système d'exploitation généraliste supportant les machines multiprocesseur à mémoire partagée.

2.3.1.4 Isolation des utilisateurs

La virtualisation permet de confiner les utilisateurs d'un noeud de la grappe dans des machines virtuelles distinctes afin de mieux les isoler. Cela évite par exemple que, dans le cadre d'une grappe partagée entre plusieurs institutions, des utilisateurs qui ne se font pas nécessairement confiance puissent voir l'ensemble des applications en cours d'exécution sur un noeud. Par ailleurs, cela permet de mieux partitionner les ressources du noeud entre les utilisateurs puisqu'ils ne peuvent accéder qu'aux ressources qui ont été dédiées à leur machine virtuelle.

Cette possibilité revêt de plus en plus d'importance du fait de l'évolution actuelle des noeuds des grappes. En effet, l'augmentation du nombre de coeurs par noeud fait que de plus en plus d'applications ne sont pas suffisamment parallèles pour exploiter efficacement la totalité d'un noeud. De plus, on voit apparaître des noeuds de calcul hétérogènes disposant d'unités auxiliaires vectorielles bien qu'il existe encore peu d'applications hybrides capables d'exploiter à la fois le parallélisme proposé par les coeurs et par les unités vectorielles. Il est donc primordial de pouvoir exécuter des tâches soumises par des utilisateurs différents sur un même noeud.

2.3.2 Des défis à relever

En dépit de tous les avantages évoqués dans la section précédente, la virtualisation est encore peu utilisée actuellement dans les grappes de calcul. Plusieurs obstacles restent à franchir pour que la virtualisation puisse être largement adoptée dans ce cadre.

Tout d'abord, les questions de performance sont prédominantes pour les applications de calcul intensif et la virtualisation est encore considérée comme trop pénalisante. Nous avons vu que, bien que la virtualisation du processeur et de la mémoire soit aujourd'hui assez efficace, il reste toujours un certain surcoût en performance qui est difficile à quantifier puisqu'il dépend très fortement des caractéristiques de l'application ainsi que de la technique de virtualisation employée. Ce problème est encore plus marqué pour les périphériques puisque les techniques d'émulation de périphériques ne permettent pas de tirer parti des matériels les plus performants. Cela affecte tout particulièrement les applications parallèles de calcul intensif qui requièrent des réseaux rapides tels que ceux utilisés dans les grappes. Les techniques de type accès direct permettent de s'affranchir de ces problèmes de performance mais limitent fortement les bénéfices apportés par la virtualisation. Le système d'exploitation utilisé dans la machine virtuelle doit être capable de piloter les périphériques hôtes, ce qui empêche de disposer de machines virtuelles portables, et il n'est pas possible de migrer simplement une machine virtuelle à chaud. En outre, donner un accès direct au réseau physique peut causer des problèmes de sécurité.

Nous avons aussi évoqué la possibilité d'utiliser les migrations de machines virtuelles pour déplier ou replier une application sur une grappe afin d'optimiser les performances ou la consommation électrique. Pour un maximum de flexibilité, l'hyperviseur doit être capable d'ordonnancer efficacement un nombre de processeurs virtuels pouvant être supérieur au nombre de coeurs disponibles sur la machine hôte. Malheureusement, comme l'hyperviseur n'est pas au courant des décisions d'ordonnancement prises par le système d'exploitation, il lui est difficile de choisir le meilleur processeur virtuel à ordonnancer.

La consommation mémoire supplémentaire liée à la virtualisation est un autre facteur limitant car les données nécessaires au fonctionnement de l'hyperviseur, telles que les tables des pages fantômes, ne sont pas négligeables. En outre, lorsque plusieurs machines virtuelles sont hébergées sur une même machine physique, il faut additionner le coût mémoire des systèmes d'exploitation et bibliothèques chargés dans chaque machine virtuelle. Cela est préoccupant puisque l'on tend actuellement vers une diminution de la quantité de mémoire disponible par coeur.

De même, le stockage disque des images des machines virtuelles de chaque utilisateur est coûteux. Ce phénomène est exacerbé dans le cadre du calcul intensif puisque ce sont des grappes de machines virtuelles qui doivent être utilisées pour exécuter des applications parallèles. Il est donc beaucoup trop encombrant de stocker séparément les images disque de chaque machine composant ces grappes virtuelles.

Enfin, la flexibilité offerte par la virtualisation, qui permet à chaque utilisateur de personnaliser son environnement d'exécution, est difficile à gérer en pratique. Tous les utilisateurs n'ont pas la compétence pour administrer leur propre grappe de machines virtuelles et cette étape est de toute façon trop fastidieuse pour être effectuée manuellement. De nouveaux outils doivent donc être développés afin, d'une part, de permettre aux utilisateurs de choisir entre différents environnements d'exécution standards et facilement configurables, et d'autre part, de stocker efficacement les images disque résultantes.

Chapitre 3

Conception d'un support exécutif basé sur la virtualisation

Sommaire

Nous avons vu que les bénéfices liés à la virtualisation sont intéressants dans le cadre du calcul haute performance, mais que des difficultés de mise en oeuvre ainsi que des problèmes de performances limitent actuellement son adoption. Dans ce chapitre, nous proposons une solution apportant plus de transparence et d'efficacité dans l'utilisation de la virtualisation pour l'exécution d'applications parallèles sur grappes. Nous commençons par décrire l'architecture générale de notre solution, puis nous détaillons les différents mécanismes mis en jeu.

3.1 Architecture générale de la solution proposée

Notre souhaitons pouvoir profiter plus simplement et plus efficacement des bénéfices apportés par la virtualisation pour l'exécution d'applications parallèles. Nous allons maintenant détailler cet objectif afin de spécifier nos principaux axes de travail.

3.1.1 Objectifs

L'utilisation de la virtualisation apporte une solution transparente à de nombreux problèmes rencontrés actuellement dans l'exploitation de grappes pour le calcul intensif. Elle permet notamment d'améliorer la tolérance aux pannes, l'équilibrage de charge, l'isolation des utilisateurs, la portabilité et la pérennité des applications et offre une flexibilité nouvelle dans l'utilisation et le débuggage de codes de niveau système personnalisés. Cependant, elle est encore peu utilisée pour les applications parallèles. D'une part à cause des problèmes de performances liés aux communications entre les machines virtuelles qui doivent passer par un périphérique virtuel émulé pour conserver la flexibilité apportée par la virtualisation. D'autre part à cause de la difficulté de configurer, instancier et maintenir une grappe de machines virtuelles ainsi que des coûts en temps, en mémoire vive et en espace de stockage qui en résultent. Dans cette section, nous décrivons comment nous souhaitons nous affranchir des contraintes liées à la virtualisation afin de pouvoir bénéficier le plus simplement et le plus efficacement possible des différents avantages qu'elle offre.

3.1.1.1 Vers une utilisation transparente de la virtualisation

Nous détaillons maintenant les objectifs à atteindre afin de rendre l'utilisation de la virtualisation la plus simple et la plus transparente possible. Ces objectifs constituent en quelque sorte le cahier des charges qui a guidé notre travail.

Intégration dans une grappe existante

L'une des conditions principales à l'adoption de la virtualisation dans les grappes est qu'elle ne perturbe pas les utilisateurs qui souhaitent déployer des applications natives. En effet, pour les utilisateurs les plus exigeants en performance, le surcoût induit par la virtualisation reste trop pénalisant, d'autant plus qu'il est difficile à prédire. En outre certaines applications implémentent déjà des solutions *ad hoc* aux problèmes résolus par la virtualisation tels que la tolérance aux pannes ou l'équilibrage de charge. La virtualisation doit donc pouvoir être utilisée à la demande et s'intégrer simplement dans le fonctionnement d'une grappe existante. Notre solution sera donc plutôt tournée vers les hyperviseurs de type II qui permettent d'héberger des machines virtuelles au-dessus d'un système d'exploitation existant.

Simplicité de déploiement

Il n'est pas souhaitable que chaque utilisateur ait à créer et administrer manuellement une grappe de machines virtuelles pour l'exécution de ses applications parallèles. Maîtriser les interfaces des différents hyperviseurs, l'installation complète et l'administration d'une machine, ainsi que la création d'un réseau virtuel, demande un investissement en temps important. En outre, puisque l'on souhaite pouvoir combiner facilement applications natives et virtualisées, il faut que les données produites dans les machines virtuelles puissent être facilement accessibles depuis la machine hôte et vice-versa.

Efficacité

Le surcoût en performance par rapport à une exécution native doit être minimisé. Puisqu'il faut démarrer une grappe de machines virtuelles à chaque lancement d'une application parallèle, on va chercher à minimiser ce temps de démarrage ainsi que le coût mémoire lié à la réplication des systèmes d'exploitation et bibliothèques dans les différentes machines virtuelles. Le surcoût en performance au niveau de l'exécution de l'application parallèle elle-même doit être limité, notamment au niveau de la virtualisation des périphériques qui impacte les communications entre les machines virtuelles.

Nous souhaitons donc pouvoir exécuter une application parallèle dans une grappe virtuelle instanciée à la volée aussi simplement et efficacement que si l'application était exécutée directement sur la machine hôte, tout en bénéficiant des avantages apportés par la virtualisation

3.1.1.2 Modèle de programmation parallèle visé

Nous nous intéressons principalement à l'exécution efficace d'applications parallélisées selon le modèle de programmation par passage de message. Outre le fait que c'est le modèle de programmation parallèle le plus utilisé actuellement pour les applications scientifiques, c'est aussi celui qui nous semble le mieux adapté à une exécution virtualisée.

En effet, comme nous l'avons vu précédemment c'est un modèle de programmation très polyvalent qui permet d'exploiter efficacement tout type d'architecture matérielle. Une application ainsi parallélisée pourra donc être déployée efficacement dans des machines virtuelles sur tout type de grappe, quelles que soient les caractéristiques des noeuds, notamment du point de vue des effets NUMA ou de la cohérence de cache. Le placement de la mémoire de chaque machine virtuelle peut alors être effectué efficacement par l'hyperviseur en fonction de ces caractéristiques afin notamment de respecter les contraintes de localité mémoire.

Utilisation de machines virtuelles mono-processeur

Nous étudions plus particulièrement le cas où chaque machine virtuelle héberge une unique tâche de type MPI. Cela permet en effet une flexibilité maximale dans l'utilisation de la virtualisation puisqu'il est alors possible de migrer indépendamment chaque tâche sur la grappe. On peut par exemple optimiser finement le placement des tâches selon différents critères ou utiliser sans restriction n'importe quel noeud de la grappe dans le cas de grappes hétérogènes dont tous les noeuds ne disposent pas du même nombre de coeurs.

En outre, on évite ainsi l'utilisation de machines virtuelles multiprocesseur qui ont un surcoût plus important [68, 69]. Nous avons en effet vu que la virtualisation de l'unité de gestion mémoire est plus coûteuse lorsqu'il y a plusieurs processeurs virtuels puisque certaines opérations doivent être sérialisées. Il est par ailleurs difficile d'ordonnancer efficacement les différents processeurs virtuels puisque lorsqu'un processeur virtuel ayant acquis un verrou est préempté par l'hyperviseur, tous les autres processeurs virtuels essayant d'acquérir le même verrou sont retardés. Cette situation est particulièrement fréquente lorsqu'il y a plus de processeurs virtuels que de processeurs physiques.

Avec des machines virtuelles mono-processeur exécutant chacune une tâche d'une application parallèle, ces problèmes d'ordonnancement sont grandement simplifiés puisque les seules dépendances entre les tâches, et donc entre les processeurs virtuels, sont exprimées sous forme de communications. Puisque ces communications font intervenir l'hyperviseur, il peut les prendre en compte pour ordonnancer efficacement les machines virtuelles, même dans le cas où il y a plus de tâches que de coeurs. Ce choix nous prive cependant de la possibilité d'utiliser des modèles de programmation hybrides. Comme nous l'avons vu précédemment, ces modèles de programmation sont encore peu utilisés mais sont amenés à se développer dans un futur proche. Une perspective de travail serait donc d'étudier ces problématiques d'ordonnancement de machines virtuelles multiprocesseur afin de supporter plus efficacement ces modèles de programmation.

3.1.2 Passage de messages efficace entre machines virtuelles

À l'heure actuelle, les inquiétudes concernant les pertes de performances constituent probablement le plus gros frein à l'adoption de la virtualisation dans le cadre du calcul intensif. Les techniques de virtualisation du processeur et de la mémoire évoquées précédemment font que la plupart des codes de calcul séquentiels peuvent être exécutés dans des machines virtuelles avec un surcoût négligeable. Cependant, pour un code parallèle de type MPI, l'efficacité des communications entre les tâches rentre en jeu.

Dans le modèle d'exécution virtualisé que nous étudions, chaque tâche est hébergée dans sa propre machine virtuelle, ce qui signifie que les échanges de messages entre les tâches induisent des communications entre les machines virtuelles. Pour que l'application parallèle s'exécute efficacement, ces communications doivent être aussi performantes que si elles étaient effectuées directement sur les machines hôtes. Lors d'un échange de message entre deux tâches d'une application MPI virtualisée, deux cas de figure se présentent.

Dans le cas où les machines virtuelles sont hébergées sur deux machines physiques distinctes, elles doivent pouvoir tirer efficacement partie du réseau hôte sous-jacent à l'aide du périphérique virtuel qui leur est exposé. Nous avons vu que les périphériques virtuels Ethernet émulés en standard par la plupart des hyperviseurs induisent encore un surcoût en performance important, notamment dans le cas des réseaux rapides trouvés dans les grappes. Les techniques d'accès direct permettent de s'affranchir de ce surcoût mais ne sont pas idéales car elles limitent la flexibilité offerte par la virtualisation.

Il est aussi possible que les machines virtuelles soient co-hébergées, c'est-à-dire hébergées sur une même machine physique. Deux tâches exécutées directement sur la machine hôte elles peuvent exploiter très efficacement l'architecture à mémoire partagée sous-jacente pour communiquer. En revanche, lorsqu'elles sont virtualisées, elles ne peuvent communiquer que par le périphérique de communication exposé par l'hyperviseur. De ce fait, même si l'hyperviseur fournit un accès direct à un périphérique réseau haute performance, les communications entre

les tâches restent moins efficaces que les communications en mémoire partagée qui auraient lieu si les tâches étaient exécutées sur l'hôte.

Aussi, de nouveaux canaux de communications entre machines virtuelles doivent être développés afin de pouvoir exécuter efficacement les applications parallèles de type passage de messages dans des machines virtuelles.

3.1.2.1 Un périphérique virtuel pour le passage de message

Afin que les machines virtuelles aient accès à un moyen de communication efficace pour le passage de message, nous proposons d'introduire un périphérique virtuel de communication disposant d'une interface bas-niveau bien adaptée à la sémantique des communications MPI.

En effet, l'interface de communication proposée en standard est celle d'un périphérique Ethernet virtuel. Or, nous avons vu que les réseaux rapides utilisés dans les grappes proposent une interface bien plus riche que celle d'un périphérique Ethernet ce qui permet notamment de court-circuiter le système d'exploitation lors des communications, de limiter les copies intermédiaires et de décharger le processeur. Ainsi, indépendamment du surcoût lié à l'émulation du périphérique Ethernet, cette interface standard n'est pas suffisante pour maximiser les performances des communications sur les grappes.

Émuler l'interface d'un réseau rapide existant offrirait plus d'expressivité que l'interface d'un périphérique Ethernet mais serait complexe à réaliser. Tout d'abord, ces interfaces sont souvent propriétaires et peu documentées. En effet, même si Infiniband est un standard, il ne définit qu'un ensemble de fonctionnalités dont l'interface bas-niveau diffère selon les constructeurs. En outre, il n'est pas garanti qu'une telle interface se prête à une virtualisation efficace, puisque cela dépend avant tout de la fréquence des aller-retour déclenchés entre hôte et invité.

Aussi, l'introduction d'une interface paravirtualisée nouvelle semble nécessaire pour permettre à des machines virtuelles de s'échanger des messages efficacement. Pour exposer cette interface aux machines virtuelles de manière simple et portable, il est naturel de définir un nouveau périphérique virtuel. En effet, puisque les systèmes d'exploitation existants sont prévus pour piloter des interfaces matérielles, ils disposent des mécanismes adaptés pour les exploiter efficacement et en exporter les fonctionnalités aux applications en espace utilisateur. Le support d'un périphérique nouveau peut être ajouté de manière modulaire par l'écriture d'un pilote de périphérique dédié. De même, la plupart des hyperviseurs émulent déjà un grand nombre de périphériques et il est en général assez simple d'en ajouter un nouveau.

3.1.2.2 Des bibliothèques de communication en contexte virtualisé

Bien que nous introduisions un périphérique virtuel nouveau, nous souhaitons tout de même pouvoir exécuter une application MPI quelconque dans notre environnement sans modification. Pour cela une pile logicielle dédiée doit être utilisée dans les machines virtuelles. Elle comprend trois éléments :

1. **Le pilote de périphérique**. Ce pilote est assez basique : il prend en charge l'interface matérielle du périphérique virtuel et en exporte les fonctionnalités en espace utilisateur via l'implémentation d'appels système standard du système d'exploitation.
2. **La bibliothèque de gestion du périphérique.** À la manière de la bibliothèque MX qui permet d'exploiter les périphériques Myrinet, l'interface de notre bibliothèque fournit les primitives de base du passage de message, c'est-à-dire notamment les communications

point-à-point. Notons que cette bibliothèque implémente des communications fiables indépendamment des migrations de machines virtuelles qui sont transparentes pour les couches supérieures.

3. **Une bibliothèque MPI supportant le périphérique virtuel.** L'interface de notre bibliothèque bas-niveau étant proche de celle de MX, qui est supportée par de nombreuses implémentations MPI, ajouter le support de notre périphérique virtuel à une bibliothèque MPI existante ne devrait pas demander trop de développements. Néanmoins, afin de disposer de plus de flexibilité pour nos différents tests nous avons développé une bibliothèque MPI réduite capable de l'exploiter. Elle supporte les principales opérations point-à-point et collectives définies par la norme MPI-1 ainsi que quelques types dérivés (l'interface exacte supportée est présentée dans l'annexe A). Cela nous a permis d'expérimenter de nouvelles fonctionnalités, comme l'adaptation dynamique des schémas de communication en fonction des migrations de machines virtuelles (voir section 4.5).

3.1.3 Un support exécutif basé sur la virtualisation

Les questions de performance ne constituent qu'une partie des entraves à l'adoption de la virtualisation. Il est aussi important de pouvoir déployer simplement et efficacement une application parallèle virtualisée. Nous proposons pour cela un ensemble de fonctionnalités qui permettent de rendre cette étape la plus transparente possible. On peut alors considérer que la combinaison de ces fonctionnalités forme un support exécutif pour applications virtualisées.

Un fork pour machines virtuelles.

Pour améliorer l'efficacité de l'instanciation d'une grappe virtuelle, nous avons implémenté une opération permettant de répliquer des machines virtuelles de manière similaire à ce que réalise l'appel système fork qui est utilisé pour répliquer un processus sur les systèmes de type Unix. Les machines virtuelles créées sont identiques à la machine virtuelle mère et contiennent notamment tous les processus précédemment lancés qui continuent leur exécution normalement. La seule différence entre les machines virtuelles se situe au niveau de la valeur retournée par la fonction fork elle-même. Chaque machine virtuelle reçoit une valeur unique qui joue un rôle d'identifiant.

À la différence d'un fork standard pour processus, nous offrons la possibilité de créer un nombre arbitraire de machines virtuelles filles en une seule opération. De plus ces machines virtuelles sont automatiquement réparties sur un ensemble de machines hôtes spécifié à l'avance. On dispose donc d'un fork distribué, permettant de tirer très simplement partie d'un ensemble de ressources physiques en instanciant autant de machines virtuelles que nécessaire (voir Figure 3.1). Une idée similaire a été proposée récemment dans le cadre du projet Snowflock [70].

Tout comme pour un fork de processus, des techniques paresseuses de type copie à l'écriture permettent de minimiser les recopies mémoire et d'augmenter la vitesse de l'opération. On se contente de transférer une seule fois la mémoire de la machine virtuelle mère sur chaque hôte puis on instancie les machines virtuelles filles en effectuant une projection de type copie à l'écriture de la mémoire de la machine virtuelle mère.

Cette opération résout donc plusieurs difficultés liées au déploiement d'applications parallèles virtualisées. Elle permet de déployer très rapidement autant de machines virtuelles que nécessaire tout en minimisant le surcoût mémoire lié à la réplication des donnés communes aux machines virtuelles comme le système d'exploitation et les bibliothèques. Pour de nombreuses

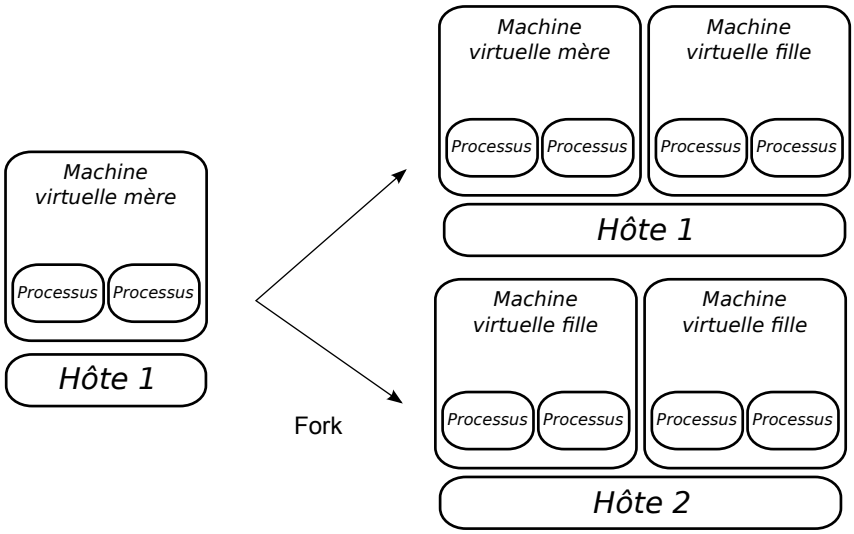


FIGURE 3.1: Réplication de machines virtuelles

applications parallèles qui effectuent une étape d'initialisation similaire sur chaque tâche, il est même possible d'effectuer le fork seulement après l'étape d'initialisation afin d'économiser temps processeur et mémoire en ne répliquant pas inutilement de données. Dans le cadre de notre bibliothèque MPI, nous effectuons le fork de machines virtuelles automatiquement lors de l'appel à la fonction d'initialisation de la bibliothèque (voir section 4.5.1). Toute zone mémoire initialisée avant la bibliothèque MPI pourra donc être partagée entre les machines virtuelles filles d'un noeud.

Accès direct aux fichiers de l'hôte

Il est peu pratique d'avoir à copier manuellement les jeux de données d'entrée et de sortie d'une application virtualisée entre le système de fichiers hôte et les disques des machines virtuelles. Nous proposons donc que les machines virtuelles puissent avoir directement accès au système de fichiers hôte, et ce avec les mêmes droits que l'utilisateur qui les a créées. Cela permet à un utilisateur de réaliser les entrées sorties de ses applications virtualisées directement dans son répertoire personnel sur la machine hôte.

Machines virtuelles légères

En allant plus loin, l'application elle-même peut résider sur la machine hôte. Cela permet d'exécuter une application en contexte virtualisé sans avoir à construire une image de machine virtuelle spécifique. Nous proposons d'utiliser une machine virtuelle embarquant un environnement d'exécution basique et un système d'exploitation simplifié, supportant uniquement l'interface para-virtuelle exposée par l'hyperviseur. Une telle machine virtuelle légère peut démarrer très rapidement et exécuter une application dont les données sont intégralement hébergées sur le système de fichiers hôte. Ces données comprennent aussi bien l'exécutable de l'application, que les divers fichiers de données nécessaires à son exécution, ou encore ses dépendances, comme les bibliothèques dynamiques.

Les mécanismes exposés ci-avant visent donc le lancement simple d'une application parallèle dans une grappe de machines virtuelles, tout en minimisant l'impact sur le temps de lancement de l'application, sur son empreinte mémoire, et sur l'efficacité des communications. Ils

permettent d'implémenter un support exécutif qui exploite les avantages apportés par la virtualisation, tel que la migration ou les points de sauvegarde transparents, et ce à moindre coût en terme de performance ou de difficulté de mise en oeuvre.

3.2 Un périphérique virtuel pour le passage de message

Cette section détaille la conception de notre périphérique virtuel dédié au passage de message. Nous commençons par passer en revue les techniques permettant d'échanger efficacement des messages entre processus exécutés de manière native, puis nous intéressons aux spécificités à prendre en compte pour les appliquer dans un environnement virtualisé, c'est-à-dire lorsque les processus sont exécutés dans des machines virtuelles distinctes. Cette analyse nous permet de discuter des fonctionnalités clés nécessaires au passage de messages efficace en contexte virtualisé qui ont guidé la conception de notre périphérique virtuel.

3.2.1 Analyse des techniques de communication natives

Nous présentons maintenant les principales techniques de communication implémentées par les bibliothèques MPI natives. Nous nous intéressons tout d'abord aux échanges de messages au sein d'un noeud à mémoire partagée, puis nous abordons le cas des communications entre noeuds reliés par un réseau rapide.

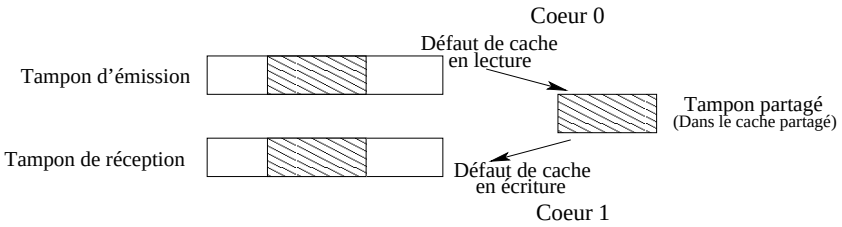
3.2.1.1 Mémoire partagée

Nous avons vu que les architectures à mémoire partagée sont à la fois de plus en plus répandues, du fait notamment de l'omniprésence des processeurs multicoeurs, et de plus en plus complexes avec l'introduction de niveaux de hiérarchie supplémentaires. De ce fait, la communauté scientifique s'est beaucoup intéressée aux techniques permettant de transférer efficacement des données sur ces architectures nouvelles. En fonction de la taille du message à échanger, un large éventail de techniques peut être mis en oeuvre. Elles s'appuient toutes sur deux mécanismes de base : une copie en deux étapes en passant par un tampon de communication partagé pré-alloué, ou une copie directe « en place » de l'émetteur au récepteur.

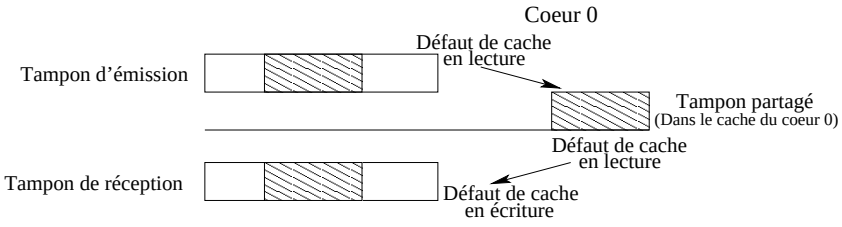
Tampon de communication intermédiaire

Bien que les processus soient exécutés dans des espaces d'adressage distincts, les systèmes d'exploitation offrent généralement différents mécanismes pour leur faire partager un segment de mémoire (par ex. *mmap*, ou les segments de mémoire partagés system V). De nombreuses implémentations MPI exploitent ces mécanismes pour créer des tampons de communication partagés. Les transferts de données entre processus se déroulent alors entièrement en espace utilisateur en écrivant et en lisant dans ces tampons de communication. Différents protocoles ont été proposés. Une possibilité est de doter chaque processus d'un tampon de réception dédie à chaque autre processus [71]. Cela permet d'éviter à deux processus envoyant simultanément un message au même destinataire de se synchroniser mais cela implique un coût en mémoire quadratique par rapport au nombre de processus. Par ailleurs, le temps de scrutation pour l'arrivée d'un nouveau message augmente linéairement avec le nombre de processus.

Afin de s'affranchir de ces deux inconvénients, il est aussi possible de n'allouer qu'un seul tampon de réception par processus et de limiter ainsi la consommation de ressources lorsque le



(a) Coeurs partageant un cache



(b) Coeurs ne partageant pas de cache

FIGURE 3.2: Impact des copies intermédiaires sur la bande passante

nombre de processus augmente. La synchronisation entre écrivains concurrents est par contre plus complexe mais peut être réalisée efficacement à l'aide de files de messages sans verrous [72].

L'utilisation d'un tampon de communication intermédiaire est particulièrement adaptée aux architectures multicoeur disposant de caches partagés, puisque si ce tampon est suffisamment petit pour tenir dans le cache, la copie intermédiaire s'effectuera en cache et aura un coût négligeable par rapport au coût de lecture et d'écriture respectivement dans la mémoire de l'émetteur et du récepteur. En revanche lorsque les caches ne sont pas partagés, une fraction importante de la bande passante mémoire est gaspillée par cette copie (voir Figure 3.2). Pour les petits messages cette technique reste néanmoins optimale car elle n'impose pas de coût particulier de démarrage ce qui permet d'atteindre de très faibles latences. Lorsque la taille des messages augmente, la copie intermédiaire devient par contre pénalisante si les coeurs ne partagent pas de cache.

Copie directe

Afin d'éviter le surcoût induit par la copie intermédiaire, il est aussi possible de réaliser une copie directe de l'émetteur au récepteur de manière à obtenir une bande passante maximale pour les gros messages. Cela demande que la copie soit effectuée dans un contexte mémoire où le tampon d'émission et de réception sont tous deux accessibles. Pour cela, une première approche consiste à effectuer la copie au sein du noyau du système d'exploitation [73, 74]. Cette approche requiert un module noyau dédié ce qui la rend moins portable. De plus, chaque communication est sujette au coût d'un appel système ainsi qu'à celui du punaisage des pages mémoire (memory pinning) ce qui restreint cette technique aux messages suffisamment gros pour que le gain de bande passante la rende rentable. Pour contourner ce problème, certaines approches emploient des threads au lieu de processus pour exécuter les tâches MPI [22, 75]. Cela permet de fournir une implémentation portable fonctionnant uniquement en espace utilisateur

mais nécessite que le code de l'application parallèle soit thread-safe. Les variables globales sont par exemple proscrites.

Ainsi, bien qu'elles soient très efficaces pour l'envoi de messages de grande taille, les techniques de copies directes sont donc encore peu répandues parmi les implémentations MPI. Nous verrons que la virtualisation permet de s'affranchir de certains problèmes rencontrés dans la mise en oeuvre de cette technique.

3.2.1.2 Réseaux rapides

Bien que des réseaux rapides divers soient utilisés dans la grappes, les bibliothèques de passage de messages s'appuient sur trois techniques de communication principales.

PIO

Le processeur émetteur peut copier de lui-même le message à envoyer, de la mémoire vers le périphérique réseau. Cette technique immobilise le processeur le temps du transfert des données et ne permet pas d'atteindre un débit très élevé. Elle reste cependant intéressante pour les petits messages car elle minimise le nombre d'opérations nécessaires au démarrage de la communication et propose donc une faible latence.

DMA pré-enregistré

Les périphériques réseau performants sont tous capables d'accéder directement à la mémoire afin d'effectuer des transferts de données plus rapides sans monopoliser le processeur pendant l'opération. Cependant, pour envoyer des données situées en espace utilisateur, la bibliothèque de communication doit d'abord punaiser les pages mémoire correspondantes, afin de récupérer leurs adresses physiques, et enregistrer ces adresses auprès du périphérique réseau. Cette étape étant très coûteuse, des tampons de communication en mémoire sont généralement enregistrés à l'initialisation. Le processeur doit donc copier le message à envoyer vers un de ces tampons de communication avant qu'il soit lu par le périphérique réseau par DMA pour être transmis. En réception, les données sont écrites en DMA vers un autre tampon pré-enregistré avant d'être copiées vers la zone de réception finale par le processeur. Cette technique permet donc d'obtenir un bon débit, notamment en pipelinant copies processeur et transferts réseau, mais la charge processeur reste importante.

Enregistrement dynamique

Pour éviter les copies intermédiaires, et décharger un maximum le processeur, il est possible d'enregistrer directement les zones mémoire d'envoi et de réception des messages. Un cache peut être utilisé pour amortir le coût d'enregistrement d'une zone mémoire sur plusieurs communications car il est fréquent que les applications parallèles réutilisent les mêmes zones pour les échanges de messages. Les communications n'impliquent alors aucune copie en mémoire, tous les transferts se faisant par DMA. Cependant émetteur et récepteur doivent se synchroniser au préalable avec un protocole de type *rendez-vous*, puisque le transfert ne peut commencer que lorsque qu'ils ont tous deux appelé des primitives d'envoi/réception concordantes. Cette technique, bien que très efficace, ne peut donc être employée que pour des messages suffisamment gros, et dans le cas ou les zones mémoire utilisées par l'application sont suffisamment statiques.

3.2.2 Contraintes supplémentaires en environnement virtualisé

En contexte virtualisé, des contraintes supplémentaires sont à prendre en compte puisque les machines virtuelles sont isolées les unes des autres, et n'ont généralement pas directement accès aux périphériques réseau hôtes. Nous étudions maintenant comment les techniques de communication vues précédemment peuvent être adaptées dans ce cadre.

3.2.2.1 Transferts en mémoire partagée

La problématique du passage de messages entre machines virtuelles sur architecture à mémoire partagée est finalement assez proche du cas de processus exécutés nativement. En effet, tout comme les processus d'un système d'exploitation, les machines virtuelles sont exécutées dans des espaces d'adressage distincts. De ce fait, deux possibilités s'offrent à nous :

– Mettre en place un ensemble de pages physiques partagées par les machines virtuelles devant communiquer, de manière à ce qu'elles puissent mettre en oeuvre, sans coût additionnel, les techniques utilisant un tampon de communication intermédiaire. Cette approche a été étudiée sur l'hyperviseur Xen en tirant partie de ses capacités de partage de pages pour mettre en oeuvre des communications par socket [76, 77] ou par MPI [78] efficaces.

– Effectuer les copies dans un espace d'adressage où à la fois les tampons d'émission et de réception sont accessibles. Deux techniques peuvent être mises en oeuvre. On peut accorder à l'une des deux machines virtuelles un droit de lecture ou d'écriture sur les pages mémoire contenant le message ou laisser effectuer l'opération par l'hyperviseur, de la même manière que cela est fait par le noyau dans le cas natif. Dans tous les cas, cela permet de n'effectuer qu'une seule copie et de profiter d'une bande passante maximale, mais impose des coûts de démarrage de communication importants, notamment s'il faut modifier une table des pages pour avoir accès aux tampons d'émission et de réception dans le même espace d'adressage. Cette technique sera donc réservée aux gros messages.

Nous sommes une fois de plus en présence d'un compromis entre latence et bande passante. Pour obtenir des performances optimales, notre solution devra sélectionner dynamiquement la méthode la plus appropriée, en fonction de la taille du message à transmettre.

3.2.2.2 Accès aux périphériques de communication

Dans le cas où les machines virtuelles sont hébergées sur des hôtes différents, elles doivent pouvoir envoyer des données par le périphérique réseau hôte. Une machine virtuelle peut manipuler directement le périphérique hôte s'il supporte la virtualisation matérielle, ou utiliser un périphérique virtuel émulé, les données étant alors réellement envoyées par l'hôte.

Accès direct

Les périphériques physiques supportant la norme SR-IOV et protégés par une IOMMU peuvent être directement manipulés par les machines virtuelles. Sur le chemin critique, l'hyperviseur n'est impliqué que lors de la délivrance des interruptions. Cela permet donc d'utiliser les techniques de communication natives en ne perdant que peu de performances.

Cependant, les contraintes importantes imposées par cette solution font qu'il est souhaitable de disposer de techniques alternatives performantes. Tout d'abord, peu de périphériques supportent actuellement les différentes normes nécessaires à une virtualisation matérielle sécurisée. Même si ce support devrait se généraliser dans les années à venir, il est intéressant de pouvoir exploiter efficacement un maximum de matériels, la virtualisation étant supposée apporter de la portabilité. En outre, la machine virtuelle doit être capable de piloter le périphérique matériel, ce qui empêche de pouvoir déployer une machine virtuelle sur n'importe quel hôte puisqu'elle doit disposer des pilotes de périphériques correspondant au matériel sous jacent. On se prive aussi de la possibilité d'utiliser des systèmes d'exploitation minimaux de type library-OS. Par ailleurs, les contraintes liées au punaisage de pages ainsi que la complexité de migrer les machines virtuelles en tenant compte de l'état du périphérique physique limitent la flexibilité apportée par la migration. Pour finir, l'hyperviseur n'étant plus impliqué dans les transferts de données, il n'est plus possible d'appliquer de politique de qualité de service ou de filtrer les paquets échangés par les machines virtuelles. Pour toutes ces raisons, nous avons choisi de ne pas utiliser d'accès directs aux périphériques hôte.

Émulation

L'utilisation d'un périphérique émulé a l'avantage et l'inconvénient de laisser l'hôte effectuer les communications sur le périphérique physique. On conserve ainsi l'abstraction du matériel apportée par la virtualisation au prix d'une perte de performance puisque des changements de contexte entre invité et hôte sont nécessaires pour chaque accès au périphérique.

Si cette contrainte impacte nécessairement la latence des échanges de message entre machines virtuelles, il est tout de même possible de tirer profit de certaines caractéristiques des réseaux rapides en passant par un périphérique émulé. En effet, mis à part le coût des changements de contexte, le problème est que les hyperviseurs émulent généralement un périphérique Ethernet dont l'interface matérielle n'est pas suffisamment expressive pour éviter les copies. À cela vient s'ajouter le surcoût lié a l'utilisation standard du protocole TCP sur ce type de réseau.

Pour mettre en oeuvre les techniques de communication vues précédemment, l'invité doit pouvoir envoyer et recevoir des données dans des tampons de communication situés en espace utilisateur sans avoir à effectuer de copie supplémentaire. Cela implique que le périphérique virtuel émulé par l'hyperviseur propose une interface de type envoi/réception de message ou RDMA adaptée afin que la destination finale des données soit connue au moment de leur réception.

3.2.3 Spécification du périphérique virtuel

Nous venons de voir qu'une interface adaptée au passage de messages entre machines virtuelles pourrait permettre d'en améliorer les performances sans avoir à fournir un accès direct au périphérique réseau hôte. Comme nous l'avons précédemment indiqué, introduire un périphérique virtuel est une bonne manière d'exposer une interface nouvelle aux systèmes d'exploitation invités. Nous présentons donc maintenant l'interface d'un périphérique virtuel simple permettant de passer des messages entre machines virtuelles.

3.2.3.1 Discussion

L'étude des techniques de passage de messages en mémoire partagée et sur réseau rapide effectuée précédemment montre que les principes mis en oeuvre dans ces deux cas sont similaires.

Pour les messages de petite taille, des tampons de communication intermédiaires prédéfinis sont utilisés. En effet, partager ces tampons de communication entre processus, ou les enregistrer auprès du périphérique réseau, sont des opérations coûteuses qui doivent être effectuées hors du chemin critique. En outre, ces tampons intermédiaires permettent d'éviter l'utilisation de rendez-vous qui synchronisent émetteur et récepteur. On minimise ainsi la latence des communications, et pour des messages suffisamment petits, l'impact des copies intermédiaires supplémentaires est faible.

Lorsque les messages sont suffisamment gros, on cherche à s'affranchir de cette copie intermédiaire en transférant directement les données de la zone d'émission à la zone de réception. Ce transfert peut prendre la forme d'une copie mémoire dans le cas où les tâches sont exécutées sur un même noeud, ou d'une communication réseau de type RDMA dans le cas contraire. Les coûts supplémentaires liés aux rendez-vous, punaisages et enregistrements mémoire sont alors rentabilisés par les gains en bande passante et en utilisation processeur.

Il est donc possible d'utiliser une interface bas-niveau commune pour le passage de messages entre tâches virtualisées, qu'elles soient exécutées dans des machines virtuelles co-hébergées ou non.

3.2.3.2 Caractéristiques générales

Le périphérique implémente des communications point-à-point entre des extrémités de communication ouvertes dans différentes machines virtuelles formant une grappe virtuelle. Chaque extrémité de communication est identifiée par une adresse unique sur toute la grappe virtuelle.

Les différents mécanismes de communications proposés sont tous fiables : une donnée envoyée est garantie d'arriver à destination. En effet, la couche de communication utilisée dans l'hôte pour émuler le périphérique virtuel sera souvent elle-même fiable. Il serait donc superflu que la gestion de la fiabilité doive être réimplémentée dans la pile logicielle de l'invité. Il est plus efficace que l'hôte se charge d'assurer la fiabilité dans les cas où le réseau sous-jacent ne la gère pas directement.

Enfin, puisque notre objectif est de connecter des machines faisant partie d'une même grappe virtuelle crée spécifiquement pour l'exécution d'une application parallèle, nous considérons que les machines interconnectées se font confiance. Aussi, aucune garantie n'est offerte contre la possibilité pour une machine virtuelle d'intercepter, corrompre, ou falsifier un flux de communication.

3.2.3.3 Interface bas-niveau

L'interface bas-niveau du périphérique propose deux mécanismes de communication qui peuvent être utilisés en fonction de la taille du message à envoyer.

Tampons de communication

Notre périphérique virtuel embarque de la mémoire et permet à l'invité d'y allouer des tampons de communication pré-définis, de taille fixe. Ces tampons peuvent être projetés dans la mémoire d'un processus invité afin que des données puissent y être écrites et lues depuis une bibliothèque de communication en espace utilisateur (voir Figure 3.3) .

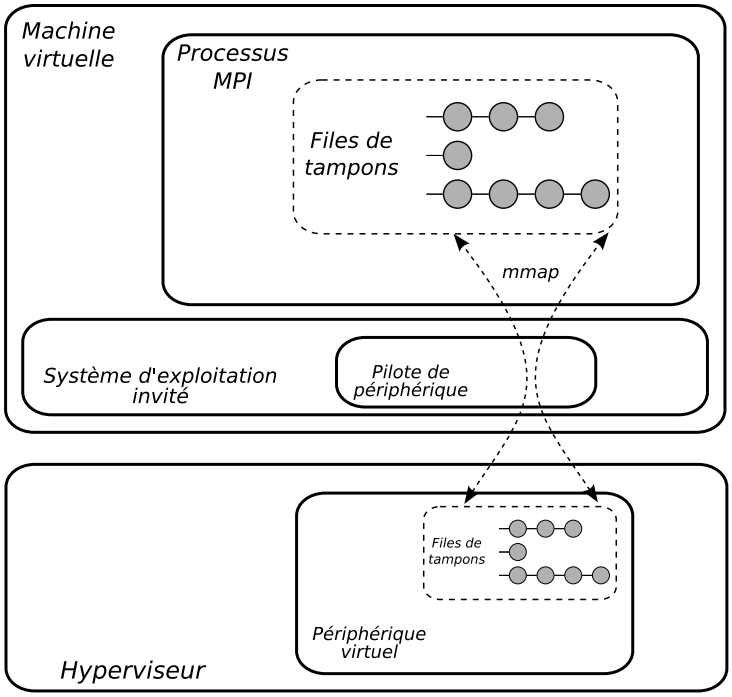


FIGURE 3.3: Projection des tampons de communication en espace utilisateur invité

Pour chaque envoi de données, la bibliothèque de communication va donc allouer autant de tampons de communication que nécessaire et y placer le message à transférer. Elle insère ensuite les tampons en question dans une file d'envoi après avoir indiqué les extrémités de communication source et destination dans l'en-tête du tampon.

Tous les tampons de communication reçus par une extrémité de communication sont placés dans une même file de réception par le périphérique virtuel. Cette file pouvant elle aussi être projetée en mémoire, il suffit, pour un processus invité, de scruter une unique adresse mémoire pour détecter l'arrivée de données. L'en-tête du tampon peut alors être lue pour déterminer l'identifiant de l'extrémité de communication source.

Ce type de communication est donc destiné à l'envoi des petits messages avec copie intermédiaire. Cette copie intermédiaire est effectuée par la bibliothèque de communication dans un tampon de communication fourni par le périphérique virtuel. Puisque ces tampons sont définis par le périphérique virtuel, et donc par l'hôte, ils peuvent être alloués dans une zone mémoire qui permet un transfert efficace vers des machines virtuelles co-hébergées ou non.

Accès mémoire distants

Le périphérique virtuel permet aussi d'effectuer des accès mémoire distants afin d'éviter les copies lors de la transmission de messages de taille plus importante.

Comme dans le cas des RDMA pratiqués par les périphériques réseau rapides, les accès mémoire distants ne peuvent s'effectuer qu'entre zones mémoire qui ont été préalablement enregistrées. L'enregistrement d'une zone mémoire consiste à fournir au périphérique virtuel une suite de couples pointeur/taille décrivant une zone potentiellement non contiguë de la mémoire physique de l'invité. Cela implique, une fois de plus, de punaiser au préalable les pages concernées afin que les projections mémoire ne changent pas en cours de transfert. Un identifiant de zone mémoire est alors associé à la zone enregistrée.

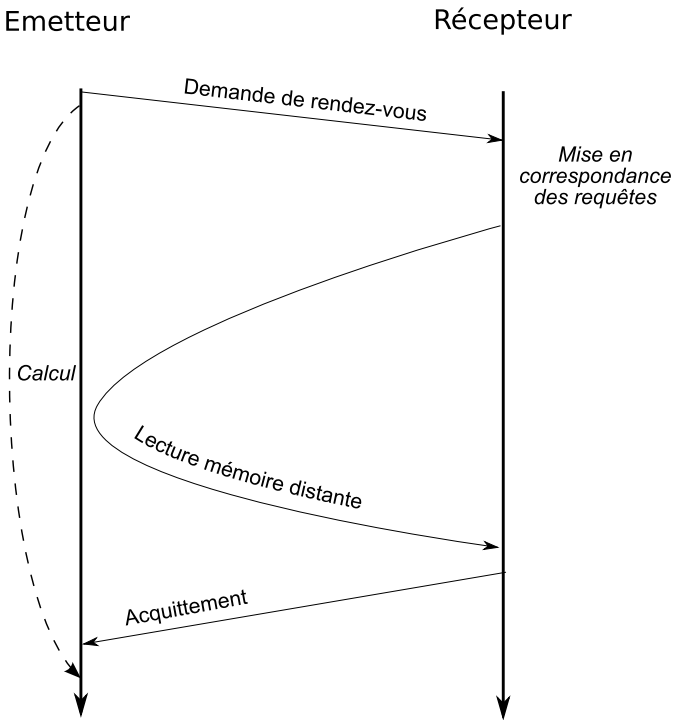


FIGURE 3.4: Utilisation d'accès distants en lecture pour recouvrir les communications par du calcul

Il est ensuite possible de déclencher une copie entre une zone mémoire distante et une zone mémoire locale, en spécifiant les identifiants des zones mémoire concernées, ainsi que l'extrémité de communication distante. La quantité de données à lire, ainsi que des décalages dans les zones mémoire locales et distantes peuvent aussi être indiqués afin de ne transférer qu'une partie d'une zone enregistrée.

Nous ne proposons que des accès mémoire distants en lecture car ils nous semblent mieux adaptés à la sémantique des communications par passage de messages que les écritures. En effet, la mise en correspondance entre requêtes d'émission et de réception de message ne peut être effectuée que du côté récepteur. Il est donc intéressant que le récepteur puisse initier le transfert de données de lui-même dès que la mise en correspondance est effectuée (voir Figure 3.4). On économise ainsi l'envoi d'un message et on augmente les possibilités de recouvrement entre calcul et communications.

Enfin, pour les deux canaux de communication, l'invité doit envoyer un signal au périphérique virtuel afin de lui indiquer que de nouvelles requêtes de communications lui ont été soumises et que les communications puissent progresser. Ce signal est implémenté de manière à déclencher une interruption logicielle afin que l'hôte puisse récupérer la main et effectuer les communications nécessaires.

3.2.3.4 Bibliothèque de passage de message

L'interface bas-niveau décrite précédemment est complexe et fastidieuse à utiliser. Aussi, elle n'est pas censée être utilisée directement et ne sert qu'à implémenter une bibliothèque de passage de messages qui abstrait les détails d'utilisation du périphérique.

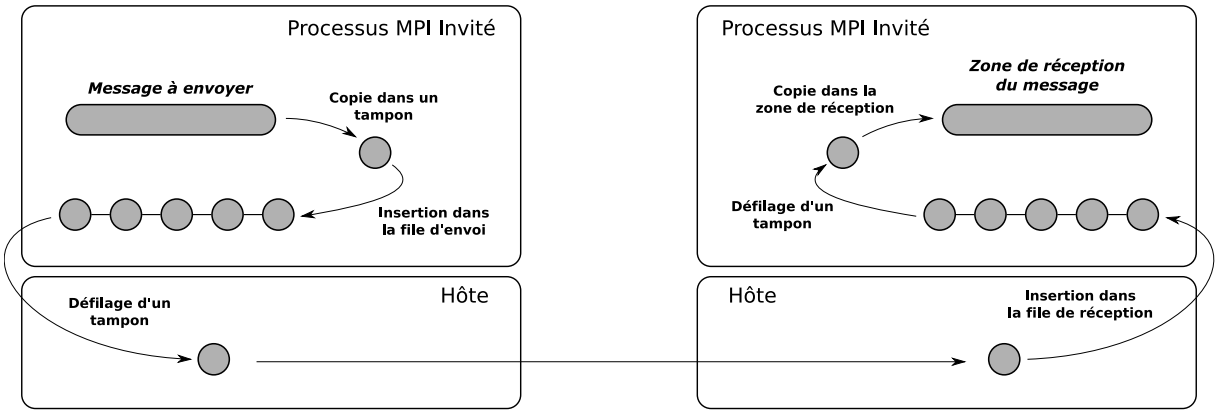


FIGURE 3.5: Transfert de message à l'aide des tampons de communication

Cette bibliothèque implémente des communications par passage de messages de type point-à-point. Pour chaque message envoyé, la bibliothèque détermine le mode de communication le plus adapté en fonction de la taille du message.

Les petits messages sont envoyés en utilisant les tampons de communication fournis par le périphérique réseau. Les messages sont découpés en autant de tampons que nécessaire et un entête décrivant le message est ajouté. Côté récepteur, si une requête de réception correspondante a déjà été postée, les donnée sont copiées directement depuis le tampon de communication vers la zone de réception du message. A l'inverse, si le message est inattendu, il est copié dans un autre tampon en attendant qu'une requête de réception correspondante soit postée.

Pour les messages plus gros, les accès mémoire distants sont utilisés. Côté émetteur, la bibliothèque de communication commence par enregistrer la zone mémoire correspondant au message à envoyer puis effectue une demande de rendez-vous. Pour cela, elle envoie un tampon de communication contenant l'en-tête du message ainsi que l'identifiant de la zone mémoire à la machine virtuelle réceptrice. Lorsqu'une requête de réception correspondant à cet en-tête est postée, la zone mémoire de réception est à son tour enregistrée et un accès mémoire distant est utilisé pour rapatrier les données. Un acquittement est ensuite envoyé à l'émetteur pour lui signaler qu'il peut libérer la zone mémoire enregistrée.

Cette bibliothèque offre donc une interface à la fois simple d'utilisation et très bien adaptée à l'implémentation d'une bibliothèque de passage de messages standard telle qu'une bibliothèque MPI, comme nous le verrons par la suite.

3.2.4 Mise en oeuvre des communications

Nous étudions maintenant comment les requêtes de communications soumises au périphérique virtuel peuvent être traitées. Nous montrons en particulier en quoi l'interface choisie permet des transferts de donnés efficaces en mémoire partagée comme sur réseau rapide et discutons de l'impact des migrations de machines virtuelles sur le traitement des communications.

3.2.4.1 Transferts de données

Chacun des deux canaux de communication proposés permettent de mettre en oeuvre les techniques natives de passage de messages étudiées précédemment.

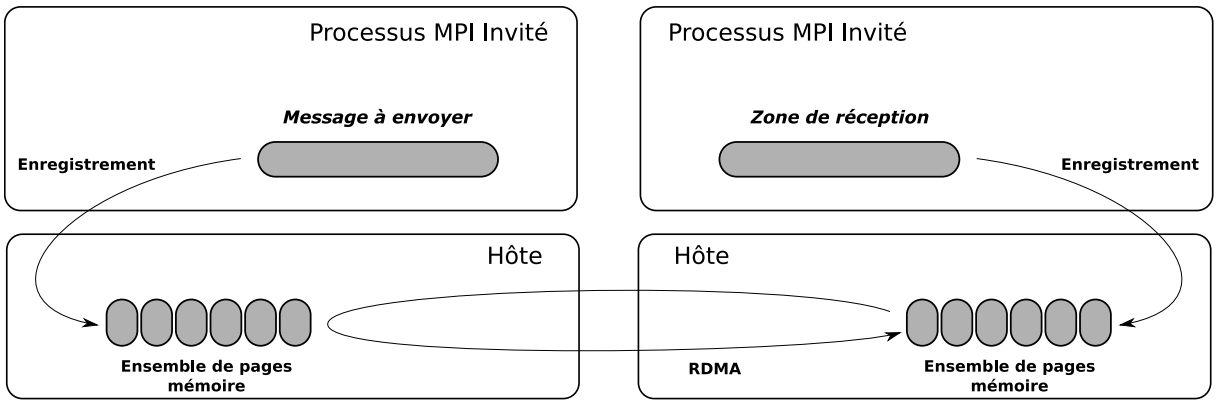


FIGURE 3.6: Transfert de message par accès mémoire distant

Tampons de communication

Une fois qu'un tampon de communication a été soumis au périphérique virtuel pour envoi, si la machine virtuelle de destination se situe sur la même machine physique, un simple échange de pointeur permet d'insérer le tampon de communication dans la file de réception. En effet, puisque l'on considère que les machines virtuelles qui communiquent se font confiance, tous les tampons de communication peuvent être partagés entre les machines virtuelles. L'échange de pointeur peut même être effectué directement dans le contexte de l'invité émetteur, sans impliquer l'hôte dans la communication.

Si les machines virtuelles sont hébergées sur des hôte différents, l'hôte hébergeant la machine virtuelle émettrice doit transférer les données sur le réseau (voir Figure 3.5). Puisque les transferts se font entre des tampons de communication prédéfinis ils peuvent être enregistrés auprès du réseau hôte pour un transfert rapide sans copie intermédiaire. Lorsqu'un hôte détecte l'arrivée de données dans un tampon de communication, il l'insère dans la bonne file de réception. Les copies supplémentaires sont donc évitées, mais le surcoût en latence induit par le changement de contexte entre invité et hôte ne peut être facilement contourné dans le cas général. Une solution consiste néanmoins à utiliser un éventuel coeur libre de l'hôte pour l'émulation du périphérique virtuel.

Accès mémoire distants

Les transferts par accès direct se font entre zones mémoire préalablement enregistrées et définies par un ensemble d'adresses physiques virtuelles. Cela permet à l'hôte d'y accéder simplement, indépendemment des translations d'adresses mises en place par l'invité. Il lui suffit de déterminer les adresses de l'hôte correspondant à ces zones mémoire et de transférer les données entre elles. Lorsque les machines virtuelles sont co-hébergées, cela revient à une simple copie mémoire. Dans le cas contraire l'hôte peut profiter de l'étape d'enregistrement virtuel pour enregistrer les zones mémoire utilisées auprès du réseau physique et transférer ensuite les données sans copies intermédiaires (voir Figure 3.6). Puisque ce canal de communication sera uniquement utilisé pour des transferts de taille importante la latence induite par les changements de contexte est moins problématique.

3.2.4.2 Cas des migrations

Migrer une machine virtuelle utilisant un périphérique Ethernet virtuel pour communiquer est relativement simple. La machine virtuelle peut simplement être détruite sur l'hôte source puis recréée dans le même état sur l'hôte de destination. Puisque le réseau Ethernet n'assure pas la fiabilité des données, les paquets qui sont envoyés à la machine virtuelle pendant la durée de la migration peuvent simplement être ignorés. Le protocole de niveau supérieur utilisé par les machines virtuelles, généralement TCP, se charge alors de retransmettre les paquets perdus.

Nos choix de conception complexifient la gestion de la migration de machines virtuelles utilisant notre périphérique, et ce à deux niveaux. Tout d'abord, puisque nous assurons la fiabilité des communications, il est de la responsabilité de l'hôte de faire en sorte que les données transmises pendant les migrations soient correctement délivrées. Par ailleurs, dans le cas des communications en mémoire partagée, les machines virtuelles peuvent être amenées à directement manipuler les files de réception de machines virtuelles co-hébergées, pour éviter les changements de contexte. Il faut donc s'assurer que les machines virtuelles ne sont pas migrées lorsqu'elles effectuent des opérations critiques afin de ne pas compromettre l'intégrité des données.

Avant chaque migration, l'hôte notifie toutes les machines virtuelles co-hébergées qu'une migration va avoir lieu, par l'intermédiaire d'une interruption déclenchée par le périphérique virtuel. Elles doivent alors interrompre leur communications directes en mémoire partagée et en informer l'hôte. Lorsque toutes les communications en mémoire partagée sont stoppées, la migration peut avoir lieu et l'hôte notifie les machines virtuelles restantes que les communications en mémoire partagée peuvent reprendre.

Pour assurer la fiabilité des transferts, nous utilisons une technique similaire, mais appliquée au niveau des communications entre les hôtes. Avant chaque migration, l'hôte source diffuse un message à tous les hôtes impliqués dans la grappe virtuelle afin de leur demander d'arrêter d'envoyer des données concernant la machine virtuelle qui va migrer. Une fois que tous les hôtes ont accusé réception de cette requête, la machine virtuelle est transférée vers son hôte de destination. Celui-ci diffuse alors la nouvelle adresse de la machine virtuelle ce qui permet aux communications la concernant de reprendre.

Cette technique à l'intérêt d'être indépendante du nombre de machines virtuelles mises en jeu, son coût dépendant uniquement du nombre d'hôtes utilisés par la grappe virtuelle. En outre, une fois qu'une machine virtuelle est prête à être migrée, on est assuré qu'il n'y a plus de données la concernant en transit. Cela signifie que l'on peut utiliser le même algorithme pour effectuer des protections reprises de grappes virtuelles. En effet, en demandant simultanément l'arrêt des communications pour toutes les machines virtuelles on s'assure qu'il n'y a plus aucune donnée en cours d'envoi sur le réseau et que l'ensemble des machines virtuelles peuvent être sauvegardées dans un état cohérent.

3.3 Bilan

Dans ce chapitre, nous avons présenté plusieurs techniques destinées à l'exécution d'applications parallèles virtualisées selon le paradigme du passage de messages. Nous proposons de les intégrer au sein d'un support exécutif capable d'embarquer efficacement chaque tâche dans sa machine virtuelle dédiée. Cet environnement permet de tirer profit de la flexibilité offerte par la virtualisation, notamment en terme de migrations de machines virtuelles, tout en minimisant les difficultés de mise en oeuvre ainsi que le coût en performance.

Pour cela, nous avons tout d'abord introduit une opération de fork pour machines virtuelles permettant d'instancier très rapidement une grappe virtuelle. En outre, nous offrons la possibilité d'exécuter des applications depuis le système de fichiers hôte, avec des machines virtuelles légères, afin de rendre l'utilisation de la virtualisation la plus transparente possible. Pour finir, nous avons proposé un périphérique virtuel nouveau, à l'interface bien adaptée au passage de messages ente machines virtuelles. L'objectif est de pouvoir exploiter aussi efficacement les architectures à mémoire partagée – lorsque les machines virtuelles sont co-hébergées – que les réseaux rapides. Nous allons maintenant présenter les principaux points d'implémentation de ces diverses fonctionnalités, ainsi que d'une bibliothèque MPI basée sur notre périphérique virtuel.

Chapitre 4

Éléments d'implémentation

Sommaire

Ce chapitre décrit l'implémentation des fonctionnalités que nous venons de proposer au sein de l'hyperviseur KVM. Nous commençons par présenter cet hyperviseur et analysons comment notre solution peut s'intégrer dans son architecture. Nous détaillons ensuite quelques points clés de l'implémentation, à savoir le concept de grappe virtuelle, la gestion du périphérique virtuel et son application à l'exécution d'applications MPI dans des machines virtuelles.

4.1 Intégration à l'hyperviseur Linux/KVM

Comme nous l'avons vu dans le chapitre précédent, nous souhaitons pouvoir déployer simplement des applications virtualisées sur une grappe existante. Nous nous sommes donc tournés vers les hyperviseurs de type II qui permettent d'exécuter des machines virtuelles au sein d'un système d'exploitation, côte à côte avec des processus standards. Puisque Linux est le système d'exploitation le plus utilisé actuellement dans les grappes, nous avons choisi de nous baser sur KVM qui est intégré en standard dans le noyau Linux depuis la version 2.6.20 sortie en 2007. Nous présentons l'architecture de cet hyperviseur, et détaillons comment notre solution s'y intègre.

4.1.1 Architecture de KVM

KVM est un module noyau permettant à Linux de jouer le rôle d'hyperviseur grâce au support matériel de la virtualisation disponible dans les processeurs récents. Ce module noyau introduit des appels système qui peuvent être utilisés par un processus en espace utilisateur pour exécuter du code dans un environnement virtualisé. Aussi, KVM lui-même n'implémente que les fonctionnalités d'un hyperviseur qui ne peuvent être effectuées en espace utilisateur, que ce soit pour des raisons de sécurité ou de performance. Un composant additionnel, exécuté en espace utilisateur, est donc nécessaire pour mettre en place un environnement virtualisé complet (voir Figure 4.1). En standard, QEMU est utilisé à cet effet. Nous détaillons maintenant les rôles respectifs joués par le module noyau et le composant en espace utilisateur.

4.1.1.1 Module noyau

Le module noyau KVM gère principalement la virtualisation du processeur et de l'unité de gestion mémoire. Il permet à n'importe quel thread utilisateur de Linux de basculer en mode virtualisé afin de faire exécuter du code par un processeur virtuel. Cependant, du point de vue de l'ordonnanceur hôte, cela reste un thread standard qui peut notamment être préempté à tout moment. Les machines virtuelles sont donc vues, par le système d'exploitation hôte, comme des processus disposant d'autant de threads que de processeurs virtuels.

À l'initialisation, le processus hôte doit spécifier la quantité de mémoire attribuée à la machine virtuelle. Pour cela, KVM permet de faire correspondre des plages de la mémoire physique de l'invité à des plages d'adresses virtuelles du processus hôte. De la mémoire doit préalablement avoir été allouée à ces adresses par les mécanismes standards d'allocation proposés par le système. Selon la classification de Goldberg, cela correspond à une f-map de type II. De cette manière, les pages mémoire physiques réellement attribuées à la machine virtuelle restent déterminées par le gestionnaire de mémoire standard de Linux et peuvent notamment être déportées sur le disque avec la même politique que celle qui est appliquée à l'ensemble des processus du système. En outre, le processus hébergeant la machine virtuelle peut simplement agir sur sa mémoire depuis le contexte hôte puisqu'elle est projetée dans son espace d'adressage.

Plusieurs threads peuvent ensuite être créés dans le processus hôte pour exécuter plusieurs processeurs virtuels simultanément. Chaque thread initialise l'état de son processeur virtuel avec les valeurs souhaitées pour ses différents registres, puis fait appel au module noyau KVM pour basculer en mode virtualisé. Le processeur virtuel démarre alors l'exécution des instructions situées au niveau de son pointeur d'instruction jusqu'à ce qu'une interruption logicielle soit déclenchée. Si l'interruption peut être traitée directement par KVM, par exemple, si elle

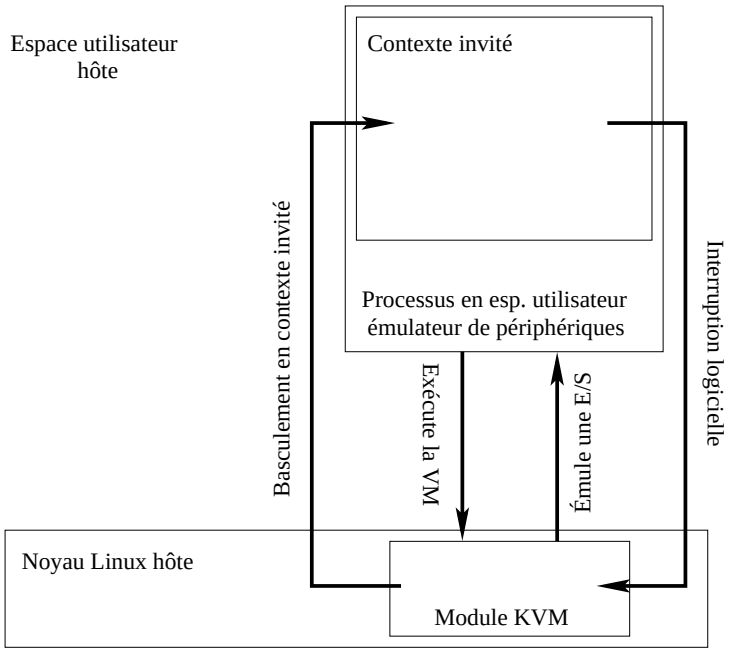


FIGURE 4.1: Architecture de l'hyperviseur Linux/KVM

concerne la MMU, l'exécution en contexte virtualisée peut reprendre directement, de manière transparente pour le processus en espace utilisateur. Dans le cas contraire, l'appel système ayant permis de basculer en mode virtualisé retourne une valeur permettant d'identifier la cause de l'interruption logicielle. C'est alors au composant en espace utilisateur d'effectuer les traitements nécessaires puis de relancer l'exécution du processeur virtuel.

En interne, KVM utilise le support matériel de la virtualisation pour basculer dans un environnement d'exécution virtualisé. Pour la virtualisation de la MMU, il tire partie du support matériel des tables des pages imbriquées lorsqu'il est disponible sur le processeur hôte et met en place des tables des pages fantômes dans le cas contraire.

4.1.1.2 Composant en espace utilisateur

Le composant en espace utilisateur s'appuie sur les fonctionnalités proposées par KVM pour émuler une machine virtuelle complète. Son rôle consiste principalement à allouer les ressources nécessaires à la machine virtuelle, comme nous l'avons vu précédemment, mais aussi à exposer les différents périphériques virtuels, que ce soient des périphériques émulés, ou des périphériques hôtes auquel un accès direct est fourni. Le composant en espace utilisateur prend aussi en charge les fonctionnalités annexes, comme la gestion des migrations de machines virtuelles. Nous étudions maintenant comment l'interface de KVM permet d'implémenter toutes ces fonctionnalités en espace utilisateur, puisque c'est sur cette interface que nous allons nous appuyer pour implémenter notre solution.

Périphériques émulés

Pour l'émulation de périphériques, le composant en espace utilisateur doit déclarer auprès de KVM les plages d'adresses physiques virtuelles correspondant à des entrées/sorties projetées en mémoire. KVM supprime alors les droits d'accès à ces plages d'adresses lorsqu'elles sont

projetées en mémoire virtuelle par le système d'exploitation invité. Chaque accès de l'invité à ces adresses déclenche donc des interruptions logicielles qui sont récupérées par KVM et relayées au processus hôte. De même, toutes les entrées/sorties par port sont systématiquement interceptées par KVM et relayées vers l'espace utilisateur hôte.

Le composant en espace utilisateur peut alors effectuer les actions nécessaires à l'émulation du périphérique comme l'envoi de paquets sur le réseau. KVM permet aussi l'injection d'interruptions dans la machine virtuelle. Elles sont alors délivrées une fois que le processeur virtuel est à nouveau exécuté.

Accès direct à un périphérique

En ce qui concerne l'accès direct aux périphériques, puisque les entrées/sorties par port sont toujours relayées au composant en espace utilisateur, celui-ci peut les transmettre telles quelles aux périphériques physiques hôtes. Par ailleurs, il peut projeter les plages mémoire d'entrées/-sorties correspondant au périphérique dans son espace d'adressage. Cela lui permet de les exposer à la machine virtuelle grâce au mécanisme qui permet de projeter des plages de ses adresses virtuelles dans une plage d'adresses physiques de l'invité. Enfin, KVM permet, d'une part de rediriger les interruptions déclenchées par un périphérique physique vers un processeur virtuel, et d'autre part d'assigner au périphérique une IOMMU permettant de traduire les adresses physiques de l'invité qu'il utilise dans ses accès DMA en adresses physiques de l'hôte.

Migrations de machines virtuelles

Pour migrer une machine virtuelle, le composant en espace utilisateur doit être capable d'extraire son état complet et de le restaurer sur une autre machine hôte. Puisque les périphériques sont gérés en espace utilisateur, KVM n'est pas impliqué dans la migration de leur état. En revanche, l'interface de KVM permet d'extraire l'état complet du processeur, ce qui permet de recréer un processeur virtuel dans le même état par la suite. En ce qui concerne la mémoire, KVM tient à jour une liste des pages qui ont été modifiées par l'invité et permet, par son interface, de lire cette liste ainsi que de la remettre à zéro. Cela permet au composant en espace utilisateur de ne transférer que les pages mémoire qui ont été modifiées. Le transfert s'effectue en lisant et en écrivant directement dans la mémoire de la machine virtuelle puisqu'elle est projetée dans l'espace d'adressage du processus hôte.

Afin de minimiser la durée pendant laquelle l'exécution d'une machine virtuelle migrée est interrompue, il est possible de poursuivre son exécution pendant le transfert de sa mémoire. Il suffit pour cela de remettre la liste des pages modifiées à zéro avant de relancer la machine virtuelle sur la machine hôte source. Une fois le transfert effectué on est en mesure de déterminer quelles sont les pages qui ont été modifiées entre-temps. La machine virtuelle ne doit alors être stoppée que pendant le transfert de ces dernières pages et son exécution peut ensuite reprendre sur l'hôte de destination.

4.1.1.3 Discussion

L'un des avantages de l'architecture de KVM est que les ressources attribuées aux machines virtuelles sont celles du processus hôte qui les crée. Le temps processeur utilisé par les machines virtuelles est celui qui est attribué aux threads exécutant leurs processeurs virtuels, et leur mémoire est allouée par le processus hôte avec des appels système standards tels que *mmap*. Cela signifie que l'interface de KVM permettant de virtualiser le processeur et la mémoire peut être

exposée à tous les utilisateurs d'une grappe sans que cela ne leur permette de monopoliser de ressources supplémentaires. En outre, les machines virtuelles sont des processus normaux qui peuvent être surveillés et contrôlés par les outils systèmes standards. Ces caractéristiques font que KVM est bien adapté à l'objectif de permettre l'exécution d'applications virtualisées au sein d'une grappe existante.

Par ailleurs, l'émulation des périphériques pouvant être effectuée en espace utilisateur, émuler un périphérique virtuel supplémentaire ne nécessite pas d'introduire de code privilégié. Cela signifie que, avec KVM, notre solution peut être implémentée entièrement en espace utilisateur, ce qui est important pour qu'elle puisse être déployée facilement. Cette flexibilité a en revanche un coût qui se répercute à chaque traitement d'une communication puisqu'il faut ajouter le coût du retour en espace utilisateur hôte au coût de changement de contexte entre hôte et invité.

Cependant, l'interface de KVM liée à l'assignation des périphériques en accès direct pose actuellement problème. En effet, nous avons vu que KVM permet d'assigner une IOMMU à un périphérique et de rediriger ses interruptions vers les processeurs d'une machine virtuelle. Cela ne suffit pas à prendre le contrôle d'un périphérique, puisqu'il faut toujours avoir les droits sur les fichiers de ressources correspondant pour y accéder, mais permet tout de même à un processus ayant accès à KVM de bloquer l'utilisation future de périphériques actuellement inutilisés. Malheureusement, l'accès à l'interface de KVM ne peut être contrôlé que de manière globale, via les droits UNIX assignés au fichier spécial correspondant à KVM. Il n'est donc pas possible, à l'heure actuelle, de permettre à un utilisateur d'instancier des machines virtuelles sans lui donner la capacité d'assigner une IOMMU à un périphérique. Cette situation est néanmoins amenée à évoluer puisque la gestion des IOMMU devrait être transférée vers un module séparé nommé VFIO (pour *Virtual Function I/O*). Il sera alors possible de donner un accès à KVM à tous les utilisateurs tout en réservant l'accès à VFIO à certains utilisateurs privilégiés.

4.1.2 Un nouveau composant en espace utilisateur

Généralement, QEMU est utilisé en espace utilisateur pour exploiter KVM. En effet, QEMU est capable d'émuler de nombreux périphériques et a donc pu être facilement adapté pour émuler des machines virtuelles complètes à partir des fonctionnalités proposées par KVM. Cependant, nous avons préféré développer un composant en espace utilisateur simple mieux adapté à nos besoins pour implémenter notre solution. Nous motivons maintenant ce choix et décrivons les caractéristiques ce nouveau composant.

4.1.2.1 Un matériel virtuel minimal

L'une des principales raisons derrière le choix de QEMU comme composant en espace utilisateur pour KVM est qu'il permet d'émuler un grand nombre de périphériques existants. Cela permet donc de simuler simplement une large palette de machines virtuelles. Cependant, dans le cadre du calcul intensif, il n'est pas nécessaire de disposer d'un tel choix de périphériques. Seuls quelques périphériques para-virtualisés sont nécessaires pour former des machines virtuelles exécutant efficacement des tâches de type MPI. Aussi, notre composant en espace utilisateur expose seulement trois périphériques :
- le périphérique de communication décrit précédemment,
- un périphérique console permettant de piloter un terminal dans la machine virtuelle depuis le terminal hôte,
- un périphérique permettant d'exporter le système de fichiers hôte dans les machines virtuelles.

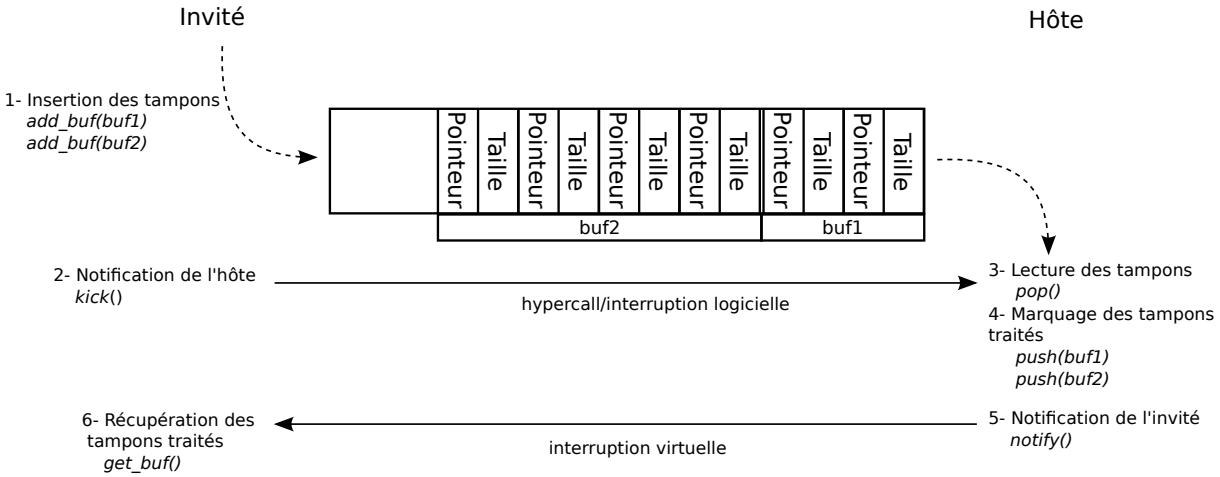


FIGURE 4.2: Files de descripteurs de tampons utilisées par VirtIO

Interface VirtIO

Nos périphériques virtuels sont basés sur l'interface VirtIO, ce qui permet de simplifier l'implémentation des pilotes de périphériques para-virtuels ainsi que de les rendre portables. En effet, VirtIO abstrait les détails spécifiques à chaque hyperviseur qui entrent en jeu dans l'implémentation d'une interface para-virtuelle, entre invité et hôte. Ainsi, une unique implémentation du pilote de périphérique s'appuyant sur l'interface VirtIO suffit, quel que soit l'hyperviseur sous-jacent .

VirtIO est basé sur des files de descripteurs de tampons permettant d'échanger des données entre hôte et invité (voir Figure 4.2). Chaque descripteur de tampon consiste en une suite de couples pointeur/taille permettant de décrire une zone non-contiguë dans la mémoire physique de l'invité. Ces descripteurs de tampons sont enfilés par la machine virtuelle et défilés par l'hôte. En retour, ce dernier notifie la machine virtuelle lorsqu'il a terminé d'effectuer les traitements correspondant à un tampon. La machine virtuelle peut choisir de recevoir une interruption lorsque ces traitements sont effectués, ou de simplement scruter le contenu d'une file dans laquelle les tampons traités sont placés.

4.1.2.2 Des grappes de machines virtuelles

Avec QEMU, les machines virtuelles sont exécutées dans des processus indépendants. Aussi, la notion de grappe de machines virtuelles n'est pas directement prise en compte. Il faut instancier manuellement autant de machines virtuelles que nécessaire et mettre en place un réseau adéquat entre elles.

Notre environnement est conçu spécifiquement pour exécuter des applications de type MPI dans des grappes de machines virtuelles pouvant être facilement instanciées à la demande grâce à une primitive de type *fork*. Aussi le composant en espace utilisateur que nous avons développé peut être distribué sur plusieurs noeuds hôtes et permet de gérer toutes les machines appartenant à une grappe virtuelle. L'utilisateur doit indiquer les noeuds physiques qu'il souhaite utiliser ce qui permet de déployer un processus par noeud. Chaque processus est alors en charge de gérer toutes les machines virtuelles hébergées sur le noeud. Lorsqu'un fork est effectué, les machines virtuelles filles résultantes sont automatiquement réparties sur les différents hôtes assignés à la grappe virtuelle et notre périphérique virtuel de communication leur offre la possibilité d'échanger efficacement des messages (voir Figure 4.3).

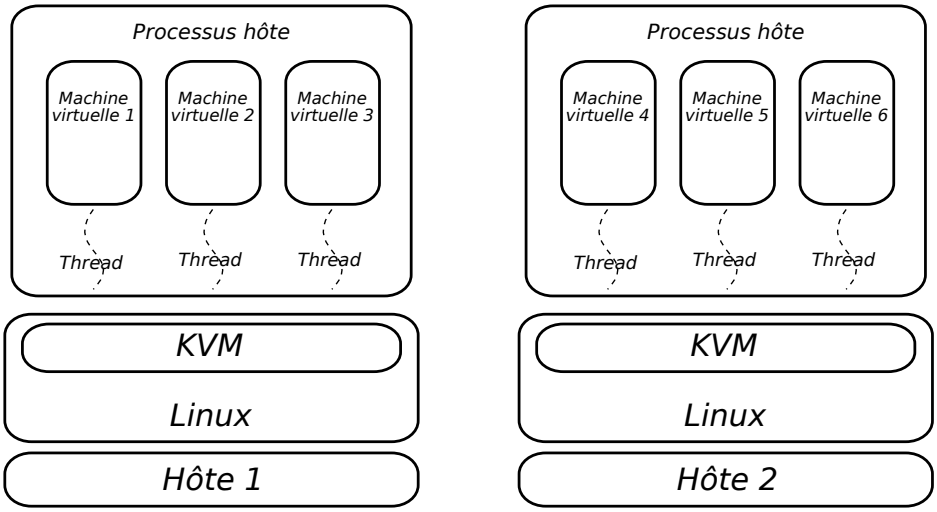


FIGURE 4.3: Gestion de grappes de machines virtuelles distribuées sur plusieurs hôtes

Cette architecture permet d'avoir une vision globale de toutes les machines virtuelles s'exécutant sur un noeud au sein d'un unique processus, ce qui offre de multiples avantages. En effet, de cette manière, la mémoire de toutes les machines virtuelles est projetée dans le même espace d'adressage ce qui permet de facilement implémenter les transferts de données directs entre machines virtuelles. En outre, on économise ainsi des ressources mémoire puisque certaines données liées à l'émulation des périphériques peuvent être partagées. Par exemple on n'instancie qu'une seule fois la bibliothèque de communication permettant d'exploiter le réseau physique hôte ce qui minimise les tampons de communication nécessaires. Pour finir, le processus hôte dispose d'une vue globale de tous les flux de communication entre les machines virtuelles d'un noeud et l'extérieur ce qui offre des opportunités d'optimisation. Cela pourrait par exemple permettre d'optimiser les flux de communication concurrents entre plusieurs machines virtuelles sur deux hôtes différents en agrégeant les paquets de données à la manière de ce que permet la bibliothèque de communication NewMadeleine.

Notre implémentation se base actuellement sur MPI pour effectuer les communications entre les différents hôtes. Cela nous permet d'exploiter simplement l'ensemble des réseaux supportés par les diverses implémentations MPI. Nous verrons cependant que cette solution n'est pas optimale en terme de performance et qu'il sera nécessaire d'utiliser directement les bibliothèques de communication bas-niveau des réseaux rapides pour en tirer la quintessence.

4.1.2.3 Un ordonnancement capable de supporter la surcharge

Nous avons vu que, dans le cas général, les coûts de virtualisation augmentent lorsqu'il y a plus de processeurs virtuels que de coeurs physiques pour les exécuter. Il est néanmoins intéressant de gérer ce cas le plus efficacement possible, notamment pour profiter de toute la flexibilité apportée par la possibilité de migrer des machines virtuelles. On peut par exemple souhaiter regrouper des machines virtuelles sur une même machine physique afin de faire de la place pour un autre calcul prioritaire, ou pour une opération de maintenance. Par ailleurs, utiliser plus de tâches MPI que de coeurs est une technique parfois utilisée afin de lisser automatiquement les déséquilibres de charge entre les tâches. De ce fait, nous avons effectué différents choix d'implémentation qui permettent de minimiser ces coûts.

Tout d'abord, comme nous l'avons dit précédemment, nous ne supportons pour l'instant que

l'utilisation de machines virtuelles mono-processeur afin de ne pas rencontrer de problèmes d'ordonnancement liés aux synchronisations entre les processeurs d'une machine virtuelle. Les seules dépendances entre les machines virtuelles d'une grappe sont déterminées par les échanges de messages à travers le périphérique virtuel de communication.

En outre, puisque les grappes de machines virtuelles instanciées sont destinées à exécuter une unique application MPI, nous pouvons les ordonnancer de manière non-préemptive en tenant compte des communications, à la manière de ce qui est effectué dans la bibliothèque MPC [22]. C'est une forme de co-ordonnancement, les machines virtuelles se passant la main lorsqu'elles attendent des communications. Cela permet de minimiser les changements de contexte entre machines virtuelles, qui sont très coûteux. Ils induisent une latence importante et diminuent l'efficacité des caches du processeur car chaque machine virtuelle travaille généralement sur des données différentes.

Pour cela, nous avons implémenté un ordonnanceur non préemptif en espace utilisateur en nous basant sur les threads ordonnancés par le noyau Linux. En effet, nous avons vu qu'avec KVM, les processeurs virtuels sont exécutés par des threads. Notre ordonnanceur utilise une file de threads prêts à être exécutés par coeur physique et demande à l'ordonnanceur noyau de restreindre l'exécution de chaque thread au coeur correspondant à sa file d'exécution. Chaque thread est par ailleurs associé à un sémaphore. A un instant donné, tous les threads sauf ceux qui sont en tête de leur file d'exécution sont bloqués sur leur sémaphores respectifs. Lorsque le thread en tête de file souhaite passer la main, il réveille le thread suivant en débloquant son sémaphore et s'endort sur le sien. On s'assure ainsi qu'à tout instant, l'ordonnanceur noyau ne peut choisir qu'un eul thread et donc une seule machine virtuelle à exécuter pour chaque coeur de la machine hôte. L'hypercall utilisé pour signaler au périphérique virtuel qu'il a des communications à effectuer (voir section 3.2.3.3) permet aux machines virtuelles de se passer la main lorsqu'elles sont en attente d'une communication. La scrutation des communications est intégrée à l'ordonnanceur et est effectuée à chaque changement de contexte.

4.2 Gestion d'une grappe virtuelle distribuée

Nous présentons maintenant quelques éléments d'implémentation relatifs à la gestion d'une grappe virtuelle distribuée sur une grappe physique. Nous expliquons comment nous avons implémenté le fork de machines virtuelles ainsi que la gestion de la topologie des grappes de machines virtuelles.

4.2.1 Fork de machines virtuelles

Conceptuellement, notre opération de fork pour machine virtuelle revient à effectuer des migrations lors desquelles la machine virtuelle d'origine n'est pas détruite. Cependant nous souhaitons en outre que cette opération soit efficace, aussi bien en terme de temps d'exécution que d'occupation mémoire.

Aussi, lorsque la machine virtuelle mère effectue un fork, on commence par créer un patron contenant les pages mémoire modifiées par la machine virtuelle, ainsi que l'état de son processeur et de ses périphériques virtuels. On diffuse ensuite ce patron à chaque hôte, et on répartit équitablement les machines virtuelles à instancier entre eux. Les données ne transitent donc qu'une seule fois sur le réseau pour chaque hôte.

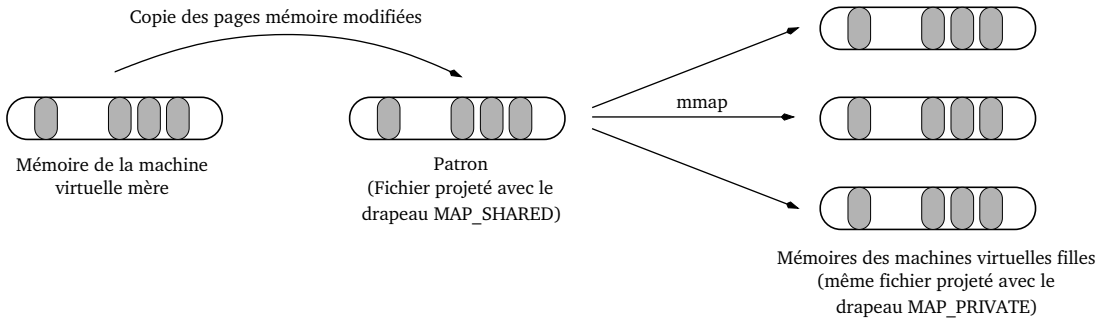


FIGURE 4.4: Implémentation d'une copie à l'écriture en espace utilisateur

En outre, nous évitons les copies mémoire intra-hôte en allouant la mémoire des machines virtuelles filles par une projection de type copie à l'écriture depuis les pages mémoire contenues dans le patron reçu. Pour effectuer cela en espace utilisateur, nous recevons les pages mémoire de la machine virtuelle mère dans un fichier projeté en mémoire par l'appel système *mmap* avec le drapeau MAP_SHARED. L'utilisation de ce drapeau permet que le contenu des pages ainsi reçues soit visible depuis les projections ultérieures du même fichier. On peut donc allouer la mémoire des machines virtuelles filles en effectuant un nouveau *mmap* du même fichier, mais avec cette fois le drapeau MAP_PRIVATE, ce qui fait que chaque machine virtuelle fille travaille par la suite sur ses propres données sans que cela n'affecte les autres (voir Figure 4.4). Puisque Linux implémente les projections de type MAP_PRIVATE par un mécanisme de copie à l'écriture, on évite ainsi les recopies en mémoire inutiles.

4.2.2 Topologie de la grappe virtuelle et migrations

Lorsque des machines virtuelle situées sur des hôtes différents se passent des messages, ce sont leurs hôtes respectifs que se chargent de relayer les données mises en jeu. Chaque hôte doit donc pouvoir déterminer la topologie de la grappe de machines virtuelles, c'est-à-dire déterminer quel hôte héberge chaque machine virtuelle. Comme cette opération est effectuée à chaque transfert de message, elle doit être très efficace pour ne pas pénaliser la latence des communications. La difficulté vient du fait que cette topologie est dynamique, puisqu'elle évolue au fil des migrations de machines virtuelles.

Nous utilisons donc un système de comptage de référence local à chaque hôte afin de minimiser le coût sur le chemin critique. L'inconvénient est que des opérations supplémentaires doivent être effectuées lors des migrations ce qui est un moindre mal, puisque c'est un évènement comparativement peu fréquent et de toute façon assez coûteux.

Lorsqu'une machine virtuelle est instanciée, l'hôte qui l'héberge diffuse sa position à tous les autres qui peuvent la stocker. Chaque hôte dispose donc localement de l'ensemble de la topologie. Par la suite, à chaque fois qu'un hôte utilise la position d'une machine virtuelle, il incrémente, pendant la durée de l'opération, un compteur de référence local associé à la machine virtuelle en question. Bien qu'il soit local, ce compteur de référence permet de s'assurer que la position de la machine virtuelle reste valide pendant l'ensemble de l'opération.

En effet, avant chaque migration, l'hôte source diffuse un message à tous les hôtes leur demandant d'invalider la position de la machine virtuelle qui migre et attend un acquittement avant de procéder à la migration. À la réception d'une demande d'invalidation, chaque hôte attend que le compteur de référence associé à la machine virtuelle tombe à zéro avant d'envoyer l'acquittement. En outre, les threads utilisant ultérieurement la position de la machine virtuelle qui

migre sont endormis le temps de la migration. Une fois la migration effectuée, l'hôte de destination diffuse la nouvelle position de la machine virtuelle, et les communications la concernant peuvent reprendre.

4.3 Périphérique de communication

Dans cette section, nous nous penchons sur quelques points clés de l'implémentation de notre périphérique virtuel de communication. Nous détaillons la gestion des communications par tampon et par copie directe, ainsi que le déroulement des migrations.

4.3.1 Tampons de communication

Nous avons vu que, pour les petits messages, notre périphérique virtuel met à disposition des tampons de communication qui peuvent être transférés entre machines virtuelles. Afin que ces transferts soient efficaces en mémoire partagée, ces tampons sont présentés dans des files sans-verrou au fonctionnement similaire à celles utilisées par le canal de communication Nemesis [72], intégré à MPICH2. Les caractéristiques de ces files font que c'est actuellement le mécanisme le plus efficace pour le passage de messages en mémoire partagée.

En effet, les files sans-verrou de Nemesis peuvent être utilisées de manière concurrente par des écrivains multiples et par un unique lecteur grâce à des opérations atomiques de type compare-and-swap. Cela permet de n'avoir qu'une seule file de réception par extrémité de communication et minimise ainsi le temps de scrutation ainsi que la consommation mémoire. En outre, la disposition des pointeurs de tête et de queue de file est effectuée intelligemment de manière à limiter le partage de lignes de caches entre lecteurs et écrivains, notamment grâce à l'utilisation d'un cache de pointeur de tête côté lecteur.

Nous associons trois files sans-verrou à chaque extrémité de communication : une file de tampons libres, une file de tampons reçus, et une file de tampons à envoyer. Un tableau dans la mémoire du périphérique virtuel permet d'indiquer aux machines virtuelles exécutées sur un noeud les identifiants des files qui sont allouées pour chaque extrémité de communication. Ces files sont projetées en espace utilisateur par le pilote de périphérique invité et sont accessibles directement par la bibliothèque de communication hôte. Le chaînage des éléments des files est basé sur des décalages, plutôt que sur des pointeurs, de manière à ce que les files puissent être projetées à des adresses mémoire différentes. Les processus qui communiquent peuvent ainsi utiliser l'appel système *mmap* implémenté par le pilote de périphérique pour projeter ces files dans leur espace mémoire.

Pour envoyer un message, la bibliothèque de communication va donc défiler un tampon depuis sa file de tampons libres et y copier le contenu du message. Si l'extrémité de communication de destination dispose d'une file locale, le tampon est directement enfilé dans la file de réception correspondante. Sinon la file d'envoi de l'extrémité de communication source est utilisée.

Lors des changements de contexte entre machines virtuelles, le mécanisme de scrutation intégré à l'ordonnanceur hôte permet de récupérer tous les tampons insérés dans des files d'envoi. L'hôte hébergeant la machine virtuelle de destination est déterminé en consultant la topologie de la grappe, et le tampon de communication lui est envoyé par le réseau. Le même mécanisme de scrutation permet de détecter l'arrivée de données sur le réseau. Elles sont directement reçues dans des tampons de communication situés dans la mémoire du périphérique virtuel qui

peuvent être insérés dans les files de réception des extrémités de communication de destination. Une fois qu'un tampon de communication est utilisé, que ce soit par l'hôte, lorsqu'il a terminé de le transférer sur le réseau, ou par une machine virtuelle, une fois qu'elle a lu son contenu, il est replacé dans sa file de tampons libres d'origine.

Dans le cadre de notre implémentation basée sur MPI, dès qu'un tampon de communication de l'hôte est libéré, une requête de réception le concernant est postée sur le réseau. Toutes les réceptions utilisent un unique tag et utilisent le drapeau MPI_ANY_SOURCE ce qui permet de les mettre en correspondance avec des envois venant de n'importe quel hôte. Tous les tampons à envoyer sur le réseau sont donc émis avec ce tag. Enfin, à chaque terminaison d'une requête de réception, l'en-tête du tampon est lue par l'hôte afin de déterminer dans quelle file de réception le placer.

4.3.2 Copies directes

Les transferts par copie directe s'effectuent en deux temps. Les zones mémoire mises en jeu doivent d'abord être enregistrées avant qu'une opération de type RDMA puisse être déclenchée. Nous étudions maintenant l'implémentation de ces deux mécanismes ainsi que d'un cache d'enregistrement permettant d'en améliorer les performances.

4.3.2.1 Enregistrement de zones mémoire

Pour effectuer un transfert direct entre deux zones mémoire en espace utilisateur, il faut s'assurer que les pages physiques associées restent les mêmes pendant la durée du transfert et enregistrer ces pages auprès du périphérique virtuel.

L'enregistrement des zones mémoire est effectué dans le pilote de périphérique virtuel, par le biais de l'interface VirtIO. Les files de descripteurs proposées par cette interface sont utilisées pour transmettre à l'hôte l'ensemble des pages mémoire correspondant à la zone à enregistrer. L'hôte assigne alors un identifiant à la zone mémoire et le retourne au pilote de périphérique. Pour le désenregistrement, le périphérique virtuel transfère simplement à l'hôte l'identifiant de la zone à laquelle l'accès est révoqué.

Afin de minimiser les coûts liés aux enregistrements, ces opérations sont exposées en espace utilisateur à travers un cache d'enregistrement implémenté par le pilote de périphérique. L'interface du pilote permet, grâce l'appel système *ioctl*, de récupérer l'identifiant d'une zone contiguë en mémoire virtuelle. Cet identifiant est garanti de rester valide jusqu'à ce qu'il soit libéré par un appel à une fonction dédiée.

En interne, le pilote punaise les pages correspondant aux zones de mémoire virtuelle qui lui sont soumises, les enregistre auprès du périphérique virtuel et retourne l'identifiant correspondant. Cependant, lorsqu'une zone est libérée, elle n'est pas désenregistrée immédiatement. On forme ainsi un cache d'enregistrement indexé par les adresses virtuelles du processus utilisant l'extrémité de communication. Si la même zone est réutilisée par la suite, son identifiant peut alors être directement retourné depuis le cache, sans avoir à enregistrer à nouveau les pages correspondantes. Deux événements peuvent conduire à une évincer une entrée du cache.

– Si trop de zones mémoire sont gardées en cache, la zone la moins récemment utilisée est désenregistrée.
– Si une zone mémoire est désallouée en espace utilisateur, ou si le noyau souhaite déporter des pages d'une zone mémoire sur disque, l'entrée du cache devient invalide puisque

l'association entre adresses virtuelles et pages physiques change. Ce cas a longtemps été difficile à gérer correctement car il fallait intercepter en espace utilisateur toutes les opérations conduisant à un changement des projections mémoire. Le noyau Linux intègre désormais les *mmu_notifiers* qui permettent à un pilote de périphérique de surveiller une zone mémoire et d'être notifié lorsque des changements de projections mémoire liés à cette zone se produisent. On utilise donc ces notifications pour désenregistrer les zones mémoire correspondantes et les supprimer du cache.

4.3.2.2 RDMA

Une fois que des zones mémoire ont été enregistrées au niveau de deux extrémités de communications, des transferts directs sous forme d'accès mémoire distants en lecture peuvent avoir lieu entre elles. Les files VirtIO sont à nouveau utilisées pour indiquer à l'hôte les opérations à effectuer. Le pilote de périphérique insère dans la file un descripteur contenant les identifiants de zones mémoire et décalages correspondant aux données locales et distantes, la quantité de données à transférer, ainsi que l'identifiant de l'extrémité de communication distante. L'appel système *ioctl* permet de déclencher cette opération depuis l'espace utilisateur.

À la réception d'une requête RDMA, si les machines virtuelles sont co-hébergées, les données peuvent directement être copiées entre les mémoire des deux machines virtuelles, puisqu'elle sont projetées dans l'espace d'adressage du composant en espace utilisateur. Dans le cas contraire, les données sont transmises sur le réseau.

Dans le cas de notre back-end MPI, les opérations se rapprochant le plus d'un RDMA sont les communications *one-sided* qui ont été introduites par la norme MPI-2. Deux opérations sont proposées, MPI_Put et MPI_Get, qui sont respectivement équivalentes à des RDMA en écriture et en lecture. Cependant la sémantique de ces communications *one-sided* est très restrictive [79], ce qui rend difficile de les utiliser pour implémenter efficacement nos RDMA. En particulier, les zones mémoires, appelées fenêtres, utilisées dans des communications *one-sided*, doivent être déclarées au préalable par une communication collective bloquante (MPI_Win_create). En outre, il est interdit de lire ou modifier localement des données d'une fenêtre pendant qu'une communication concernant la fenêtre se déroule, même si cette communication touche des données différentes. Il n'est donc pas possible de déclarer à l'avance l'ensemble de la mémoire des machines virtuelles pour éviter ces communications collectives par la suite.

Aussi, nous nous basons sur des envois de messages pour effectuer les transferts réseau, ce qui permet en outre de conserver la compatibilité avec les bibliothèques MPI ne supportant que la norme MPI-1. Pour cela, l'hôte effectuant la lecture à distance poste une requête de réception avec un tag inutilisé et demande à l'hôte distant de lui envoyer les données à lire en utilisant ce tag.

4.3.3 Migrations

Nous utilisons à nouveau les files VirtIO pour permettre à l'hôte de notifier le pilote de périphérique invité qu'une migration va avoir lieu. Le pilote de périphérique doit alors s'assurer qu'il n'y a plus de communications en cours utilisant les files de tampons avant d'envoyer un acquittement à l'hôte, toujours en utilisant VirtIO. En effet, une machine virtuelle ne peut être migrée en cours de manipulation d'une file, et il faut s'assurer que toutes les machines virtuelles locales n'utilisent plus la file de réception la machine qui migre. La difficulté vient du fait que les communications par tampon s'effectuent en espace utilisateur. Il faut donc relayer la

notification de migration à la bibliothèque de communication en espace utilisateur afin qu'elle arrête ses communications.

Pour cela, le pilote de périphérique virtuel implémente un descripteur de fichier qui peut être lu afin de récupérer les événements de migration. Dans la bibliothèque de communication, ce descripteur de fichier est lu de manière bloquante par un thread dédié. Ce thread est donc réveillé lorsqu'une migration à lieu, et a la charge d'arrêter les communications. Pour cela, un mécanisme de comptage de référence lui permet de déterminer lorsque les fonctions manipulant les files de tampons sont terminées, et de bloquer les appels ultérieurs à ces fonctions. Il peut alors indiquer au pilote de périphérique que les communications sont stoppées.

Une fois la migration terminée, et la table indiquant les extrémités de communications locales mise à jour, un mécanisme similaire permet de notifier le pilote de périphérique invité, puis la bibliothèque de communication utilisateur, que les communications peuvent reprendre.

4.4 Périphériques annexes

Un minimum de périphériques supplémentaires sont nécessaires pour pouvoir interagir avec les grappes de machines virtuels que nous instancions. Nous souhaitons en particulier pouvoir piloter des terminaux dans les machines virtuelles, et partager les données situées sur le système de fichiers hôte. Nous émulons donc deux périphériques VirtIO existants, pour lesquels Linux dispose déjà de pilotes : un périphérique console et un périphérique exposant le système de fichiers hôte basé sur le protocole 9P. Nous présentons maintenant le fonctionnement général ainsi que l'implémentation de ces deux périphériques au sein de notre composant en espace utilisateur.

4.4.1 Console virtuelle

La périphérique virtuel de console permet de relier un terminal de l'hôte à un terminal de l'invité. Cela permet d'interagir de manière transparente avec un shell ou avec toute autre application virtualisée, comme si elle était exécutée depuis un terminal hôte. Ce périphérique utilise deux files VirtIO afin de pouvoir transférer des données dans les deux directions.

La première file permet de transférer des données vers l'hôte, typiquement les sorties standard des applications. Pour cela, l'invité copie les données sortantes dans des tampons qu'elle expose ensuite à l'hôte en les insérant dans cette file. À la réception de ces tampons, notre composant en espace utilisateur hôte affiche simplement leur contenu sur la sortie standard.

La seconde file concerne les données qui sont envoyées à la machine virtuelle. L'invité y insère des tampons libres qui peuvent être utilisés par l'hôte pour y placer ses données. Notre composant en espace utilisateur récupère donc les entrées brutes qui lui sont soumises par son terminal de contrôle et les copie dans ces tampons. Le pilote de périphérique dans l'invité injecte alors ces données dans un terminal afin qu'elles puissent être lues par l'entrée standard d'une application.

4.4.2 Export du système de fichiers hôte

Pour exporter le système de fichiers hôte dans la machine virtuelle, nous utilisons un périphérique virtuel conçu pour relayer le protocole 9P entre invité et hôte. Le protocole 9P est

un protocole réseau permettant d'interagir avec un système de fichiers distribué. De manière générale, c'est donc un protocole similaire au protocole NFS généralement utilisé pour cette tâche. Le protocole 9P a cependant été conçu pour être à la fois plus simple et plus générique que les protocoles de systèmes de fichier distribués existants.

En effet, ce protocole est indépendant du protocole réseau sous-jacent utilisé pour transférer les données. En outre, il se base sur un mécanisme simple de requêtes/réponses entre client et serveur. Seules 13 requêtes différentes sont définies et correspondent directement aux principales fonctions de manipulation de fichier (parcours de l'arborescence, ouverture, fermeture, création, suppression, lecture, écriture, manipulation des méta-données...).

Ces caractéristiques font que ce protocole est bien adapté à la virtualisation d'un système de fichiers et le noyau Linux propose désormais en standard un client 9P utilisant l'interface VirtIO pour transmettre ses requêtes. Notre composant en espace utilisateur contient donc un serveur 9P qui se charge d'exporter le système de fichiers hôte. Ce serveur est assez simple puisque la plupart des requêtes 9P correspondent directement à des opérations sur le système de fichiers hôte.

4.5 Bibliothèque VMPI

Afin de pouvoir virtualiser des applications existantes le plus simplement possible, nous avons implémenté une bibliothèque MPI basée sur notre périphérique virtuel. L'idée est de pouvoir lancer une application parallèle MPI dans une grappe de machines virtuelles aussi simplement que sur une grappe physique. Dans cette section, nous présentons le principe de fonctionnement de cette bibliothèque et détaillons la gestion des communications collectives en fonction de la topologie de la grappe virtuelle.

4.5.1 Principe de fonctionnement

Avec VMPI, l'instanciation des machines virtuelles exécutant l'application parallèle est transparente pour l'utilisateur. Le lanceur associé à notre bibliothèque exécute la première instance de l'application dans une machine virtuelle légère contenant un noyau Linux minimal. Les fichiers de l'application sont lus et écrits directement sur le système de fichiers hôte par le périphérique virtuel 9P, et le périphérique virtuel de console permet de récupérer la sortie standard de l'application et de la rediriger sur le terminal de l'hôte.

Lorsque l'application effectue un appel à MPI_Init, qui doit nécessairement être exécuté avant tout autre appel MPI, notre bibliothèque déclenche un fork afin de créer une grappe virtuelle dimensionnée en fonction du nombre de tâches MPI spécifié au lanceur. Cela offre l'avantage que, si l'application initialise certaines données avant d'utiliser la bibliothèque MPI, elles pourront être partagées en mémoire grâce au mécanisme de copie à l'écriture utilisé par le fork. Les hôtes utilisés pour l'exécution de la grappe virtuelle peuvent être spécifiés au lanceur dans un fichier similaire à ceux utilisés par les bibliothèques MPI standard.

4.5.2 Communications collectives

Les communications point-à-point se résumant généralement à un simple appel à la bibliothèque de gestion du périphérique virtuel, l'essentiel du code de la bibliothèque VMPI concerne l'implémentation des communications collectives.

Nous avons choisi de les implémenter en nous basant sur des communications point-à-point. En effet, bien que des algorithmes spécifiques soient parfois plus efficaces en mémoire partagée, leurs performances dépendent souvent de caractéristiques fines du matériel sous-jacent. Il est donc difficile de les optimiser dans le cas général. En outre, ce type d'algorithme pose des contraintes supplémentaires dans un environnement virtualisé puisqu'il faut tenir en compte des migrations de machines virtuelles.

Cependant, il est tout de même nécessaire d'adapter les schémas de communication utilisés en fonction de la topologie de la grappe virtuelle de manière à effectuer un maximum de communications en mémoire partagée. Ces dernières sont en effet beaucoup plus efficaces que les communications sur le réseau, notamment lorsque plusieurs tâches utilisent simultanément le même périphérique réseau.

4.5.2.1 Modélisation de la hiérarchie des communications

Afin de maximiser le ratio entre utilisation de la mémoire partagée et du réseau, notre bibliothèque MPI gère les communications collectives de manière hiérarchique. Pour cela, chaque communicateur MPI est décomposé en un arbre qui trie les tâches appartenant au communicateur en fonction de leur position dans la grappe. Nous utilisons actuellement trois niveaux de hiérarchie : le niveau global, qui regroupe toutes les tâches, le niveau noeud, qui regroupe les tâches communiquant en mémoire partagée, et le niveau coeur, qui regroupe les tâches se partageant un même coeur. Notons que l'arbre pourrait facilement être étendu pour prendre en compte des niveaux de hiérarchie supplémentaires, par exemple dans le cas où les machines virtuelles sont exécutées sur une grappe de grappes.

La construction de l'arbre est effectuée à la création d'un communicateur. Nous avons ajouté un hypercall permettant à chaque machine virtuelle de récupérer l'identifiant de l'hôte sur lequel elle est actuellement hébergée ainsi que le numéro du coeur qui l'exécute. Les tâches d'un communicateur peuvent alors s'échanger cette donnée afin que chacune puisse construire l'arbre complet. On s'assure ainsi que toutes les tâches ont une vision cohérente de la topologie de la grappe virtuelle indépendemment de toute migration concurrente.

L'arbre doit cependant évoluer au fil des migrations afin refléter la topologie réelle de la grappe. Une solution simple consiste à le reconstruire au bout d'un certain nombre d'utilisations d'un communicateur pour des opérations collective. En effet, chaque opération collective est un point de synchronisation pour toutes les tâches du communicateur. On s'assure ainsi de façon simple que toutes les tâches mettent à jour leur arbre au même moment.

4.5.2.2 Application à des communications collectives hiérarchiques

Les communications collectives peuvent être effectuées efficacement en les décomposant selon les niveaux de l'arbre. Étudions par exemple les cas d'une diffusion et d'une barrière qui illustrent différentes manières d'exploiter cet arbre pour construire un schéma de communications.

- Pour une diffusion, on procède en partant de la racine et on réalise une sous-diffusion à chaque noeud de l'arbre. Le groupe de tâches participant à cette sous-diffusion est constitué d'une tâche appartenant à chaque fils du noeud courant et inclut toujours une tâche qui a déjà reçu les données à diffuser.
- Pour une barrière, on commence par remonter l'arbre à partir des feuilles selon un principe similaire en effectuant des sous-barrières pour chaque noeud. Une fois arrivé à la racine, on effectue une diffusion pour signaler la fin de la barrière à toutes les tâches.

Notons que les communications collectives internes peuvent être implémentées différemment à chaque niveau de l'arbre. Pour les deux premiers niveaux, c'est-à-dire pour le niveau global et le niveau noeud, nous utilisons les algorithmes parallèles utilisés en standard dans les bibliothèques MPI actuelles décrites dans [80]. En revanche pour le niveau processeur, paralléliser ces communications collectives est contre-productif puisque les tâches s'exécutent les unes après les autres sur le processeur. Nous effectuons donc les communications séquentiellement afin de minimiser le nombre de changements de contexte entre machines virtuelles.

4.6 Bilan

Dans ce chapitre, nous avons décrit l'implémentation de nos propositions au-dessus de l'hyperviseur KVM. Ce choix d'hyperviseur nous a permis de développer un support exécutif s'intégrant dans une grappe Linux existante, sans avoir à introduire de code privilégié dans l'hôte. Grâce à la bibliothèque MPI que nous avons développée, VMPI, nous sommes en mesure de virtualiser de façon transparente des applications parallèles existantes. Il nous reste maintenant à évaluer les performances de cette solution.

Chapitre 5

Évaluation

Sommaire

Dans ce chapitre, nous présentons le résultat d'expériences destinées à évaluer l'efficacité de notre solution dans le cadre du calcul intensif. Nous nous focalisons dans un premier temps sur des tests élémentaires conçus spécifiquement pour mettre en évidence le coût de diverses opérations critiques concernant la gestion d'une grappe de machines virtuelles ainsi que les performances de notre périphérique de communication. Nous comparons ensuite les temps d'exécution de divers codes de calcul parallèles dans des grappes de machines virtuelles aux temps obtenus lorsque ces mêmes codes sont exécutés sur les noeuds hôtes.

5.1 Machines de test

Nous décrivons maintenant les 3 machines sur lesquelles nous avons effectué nos tests de performance.

Nehalem La machine Nehalem est notre machine de développement. C'est une machine autonome équipée de 24GiB de mémoire vive et de deux processeurs quad-core E5520 cadencés à 2.27GHz. Ces processeurs sont basés sur l'architecture Nehalem. Nous utilisons

	Durée	%
Initialisation du composant en esp. utilisateur	0.12s	18%
Initialisation de KVM	0.1s	15%
Démarrage Linux invité	0.1s	15%
Destruction de KVM	0.24s	37%
Destruction du composant en esp. utilisateur	0.08s	12%
Total	0.65s	100%

TABLE 5.1: Détail du coût d'instanciation d'une machine virtuelle légère

cette machine pour des tests nécessitant des droits d'administration sur la machine ou une version de noyau spécifique mais aussi pour mettre en avant les différences de comportement entre les architectures de processeur Nehalem et Core.

Jack Jack est une grappe de test composée de deux noeuds reliés par un réseau Infiniband. Chaque noeud est équipé de 24GiB de mémoire vive et de deux processeurs hexa-core X5650 cadencés à 2.67GHz. Ces processeurs de dernière génération sont eux aussi basés sur l'architecture Nehalem. Cette grappe nous permet de tester les performances des communications inter-noeud avec un processeur récent.

Fortoy Fortoy est une grappe composée de noeuds bi-processeurs quad-core E5462 cadencés à 2.8GHz et basés sur l'architecture Core. Chaque noeud est par ailleurs équipé de 8GiB de mémoire vive. Nous disposions de 8 noeuds de la grappe sur lesquels KVM était installé ce qui nous a permis d'exécuter des applications parallèles virtualisées sur 64 coeurs.

5.2 Micro-évaluations

Cette section met en évidence le coût des diverses opérations nécessaires à l'exécution d'un code de calcul parallèle dans une grappe de machines virtuelles. Nous nous intéressons tout d'abord au temps d'instanciation des machines virtuelles, qui impacte le temps de démarrage d'une application parallèle virtualisée. Nous détaillons ensuite les performances des communications utilisant notre périphérique virtuel et finissons par une analyse du coût des migrations de machines virtuelles.

5.2.1 Instanciation de machines virtuelles

Dans notre modèle d'exécution virtualisé, le lancement d'une application parallèle de type MPI nécessite de démarrer une première machine virtuelle mère. Cette machine virtuelle est ensuite répliquée autant de fois qu'il y a de tâches MPI grâce à notre fork de machine virtuelle. Nous analysons maintenant le coût de ces deux opérations, et distinguons le cas où le fork de machines virtuelles est effectué sur un unique noeud du cas où il est distribué sur une grappe de machines physiques.

5.2.1.1 Lancement d'une machine virtuelle Linux

Nous commençons par analyser le coût lié à l'instanciation d'une machine virtuelle pour exécuter une application. Pour cela nous mesurons le temps nécessaire au démarrage d'une machine

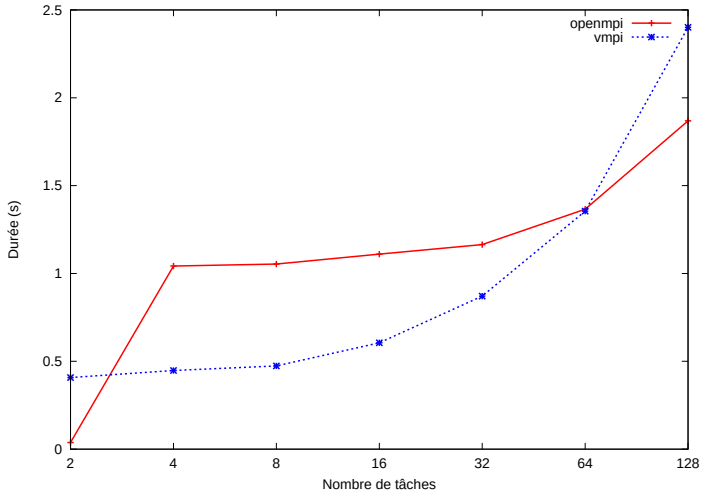


FIGURE 5.1: Coût de lancement de tâches MPI sur un noeud, en mode natif et virtualisé

virtuelle légère utilisant un noyau Linux n'intégrant que le support matériel nécessaire au pilotage de nos périphériques virtuels. Nous détruisons la machine virtuelle dès que le noyau Linux est entièrement démarré. Le tableau 5.1 décompose le temps total d'exécution puis de destruction d'une machine virtuelle. Ces tests ont été effectué sur la machine Nehalem, et les résultats présentés sont les moyennes obtenues sur 50 exécutions du test.

Nous constatons que l'instanciation et le démarrage d'une machine virtuelle légère sont très rapides. Environ 0.3s seulement sont nécessaires à l'obtention d'un environnement dans lequel des programmes en espace utilisateur peuvent s'exécuter dans une machine virtuelle. Ce temps se décompose de manière égale entre le démarrage du noyau Linux, l'initialisation d'une machine virtuelle par le module noyau KVM, et l'initialisation du composant en espace utilisateur. Cette dernière comprend notamment l'allocation de la mémoire de la machine virtuelle dans laquelle le noyau Linux et son initrd doivent être copiés.

La destruction d'une machine virtuelle est à-peu-près aussi longue que son démarrage. Elle consiste principalement en la libération des ressources allouées par le module noyau KVM qui est relativement coûteuse.

5.2.1.2 Fork

Nous étudions maintenant le coût d'instanciation d'une grappe de machines virtuelles à l'aide de notre opération de fork. L'objectif est de mesurer son impact sur le temps de lancement d'une application parallèle MPI. Notre test élémentaire consiste à exécuter une application minimale qui effectue une barrière après avoir initialisé la bibliothèque MPI et à mesurer le temps total écoulé jusqu'à l'exécution de la barrière. Nous comparons les temps obtenus avec la bibliothèque OpenMPI, en utilisant le lanceur standard, et avec notre bibliothèque VMPI. Notons que dans ce dernier cas, le temps mesuré inclut le démarrage de la machine virtuelle mère ainsi que le fork.

La figure 5.1 présente le résultat de ce test lorsque l'on fait varier le nombre de tâches sur un noeud de la grappe Fortoy. Chaque test est effectué 10 fois et le meilleur temps est conservé.

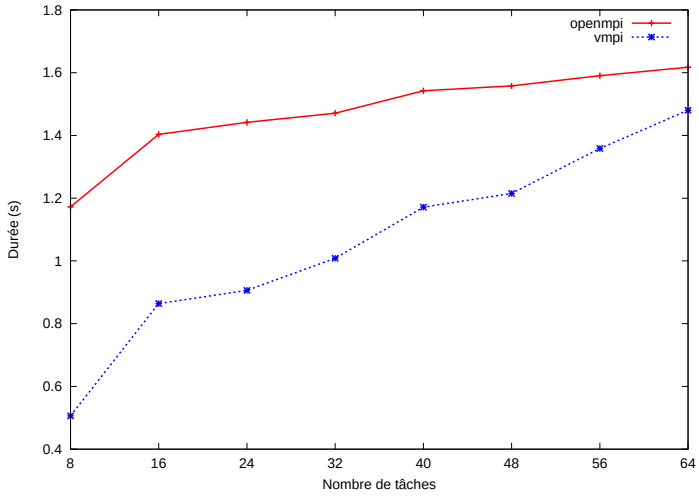


FIGURE 5.2: Coût de lancement de tâches MPI sur une grappe, en mode natif et virtualisé

On constate que les temps de lancement mesurés dans les deux cas sont similaires. Lorsque le nombre de tâches reste raisonnable devant le nombre de coeurs disponibles sur la machine - jusqu'à 8 tâches par coeur - il est même légèrement plus rapide de lancer les tâches dans des machines virtuelles avec VMPI que de les lancer sur l'hôte avec OpenMPI. Ce résultat met en évidence l'efficacité du fork de machines virtuelles.

5.2.1.3 Fork distribué

Ce test d'instanciation de machines virtuelles est similaire au précédent à la différence près que les machines virtuelles sont distribuées sur une grappe de machines physiques. Nous faisons varier le nombre de machines physiques utilisées au sein de la grappe Fortoy, et créons autant de tâches MPI que de coeur. Nous présentons les temps obtenus par OpenMPI et VMPI sur la figure 5.2. Chaque test est effectué 10 fois et le meilleur temps est conservé.

On constate que les temps de lancement des tâches MPI sont similaires dans les cas natifs et virtualisés. Cela montre que le coût de réplication des machines virtuelles sur une grappe n'est pas pénalisant. Cependant l'allure de la courbe révèle que l'implémentation de notre fork distribué devra être améliorée pour passer efficacement à l'échelle puisque son coût augmente rapidement avec le nombre de machines hôtes impliquées. En effet, actuellement, les données permettant de créer les machines virtuelles filles sont envoyées séquentiellement à tous les hôtes à partir de l'hôte hébergeant la machine virtuelle mère. Un algorithme de diffusion distribué basé sur un arbre permettrait de minimiser le temps de transfert quand le nombre d'hôtes augmente.

5.2.2 Communications

Les tests présentés dans cette section visent à mettre en évidence les performances des communications utilisant notre périphérique virtuel. Nous nous focalisons sur les performances des diverses primitives de communication MPI et utilisons pour cela la suite de tests IMB (pour

Intel MPI Benchmarks). Nous présentons tout d'abord les performances brutes de latence et de bande passante obtenues lors de communications point-à-point, puis nous nous intéressons aux communications collectives. Nous distinguons en outre, pour chaque test, le cas où toutes les communications ont lieu en mémoire partagée du cas où les machines virtuelles sont distribuées sur plusieurs noeuds de la grappe.

Les différentes valeurs reportées dans cette section sont les valeurs brutes obtenues lors d'une exécution d'IMB. En effet cette suite de tests fournit d'elle même des temps d'exécution moyennés sur un grand nombre d'opérations. Notons par ailleurs que nous exécutons ces tests avec le paramètre *-off_cache* proposé par IMB. Ce paramètre permet de faire varier les zones mémoire qui sont utilisées pour les échanges de messages afin que celles-ci ne soient pas systématiquement en cache préalablement aux transferts. On obtient ainsi des conditions de test plus proches de celles rencontrées dans des applications réelles.

5.2.2.1 Communications point-à-point

Pour évaluer les performances des communications point-à-point, nous utilisons le test classique PingPong. Dans ce test, deux tâches s'échangent tour à tour un message d'une taille donnée, et la latence reportée correspond à la moitié du temps nécessaire à un aller-retour du message. La bande passante est simplement obtenue en divisant la taille du message par la latence.

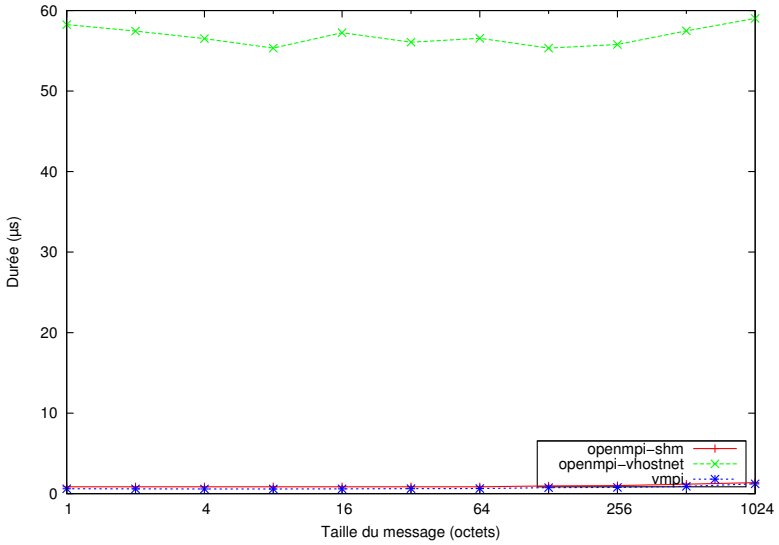
Communications en mémoire partagée

La figure 5.3 présente les résultats obtenus sur la machine Nehalem. Nous comparons les performances obtenues pour le test PingPong lorsque les tâches sont exécutées dans 3 environnements différents.

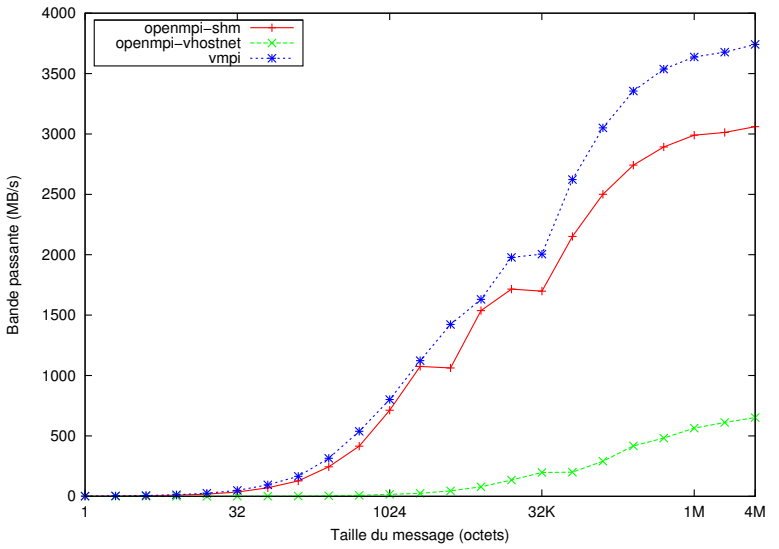
– Les courbes intitulées *openmpi-shm* correspondent à une exécution des tâches MPI sur l'hôte, avec la bibliothèque OpenMPI.
– Les courbes intitulées *openmpi-vhostnet* correspondent à l'exécution de deux tâches MPI dans deux machines virtuelles co-hébergées. Ces machines virtuelles utilisent KVM et QEMU en composant en espace utilisateur et communiquent avec le périphérique Ethernet paravirtualisé le plus efficace proposé actuellement par QEMU. Les tâches MPI utilisent dans ce cas aussi la bibliothèque OpenMPI pour exploiter le périphérique Ethernet virtuel. Notons que ce périphérique nécessite un noyau Linux hôte récent (le module vhost-net n'est apparu que dans la version 2.6.34) ainsi que les droits administratifs nécessaires à la création d'un pont réseau. Nous n'avons donc pu le tester que sur notre machine de développement
– Enfin les courbes intitulées *vmpi*, correspondent à l'exécution de tâches dans des machines virtuelles co-hébergées utilisant notre périphérique de communication et la bibliothèque VMPI.

Quelle que soit la quantité de données transférée, on constate que l'utilisation d'un périphérique Ethernet virtuel impose un surcoût très important par rapport à des solutions mettant directement à profit la mémoire partagée pour les échanges de messages. Ce résultat souligne donc la nécessité d'introduire un périphérique virtuel nouveau pour pouvoir exécuter efficacement des applications MPI dans des machines virtuelles.

Notre périphérique virtuel permet quant à lui d'atteindre des latences et bandes passantes similaires à ce que l'on obtient en exécutant les tâches sur l'hôte. Pour des messages de grande taille la bande passante atteinte est même sensiblement meilleure.



(a) Latence

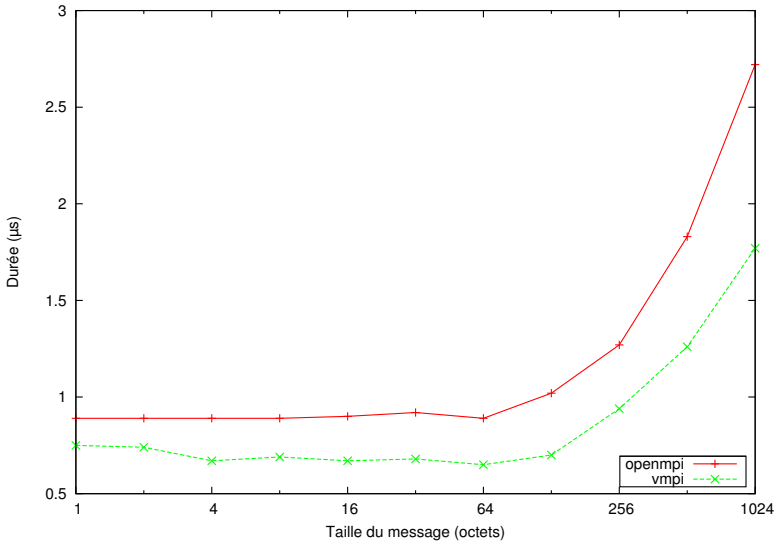


(b) Bande passante

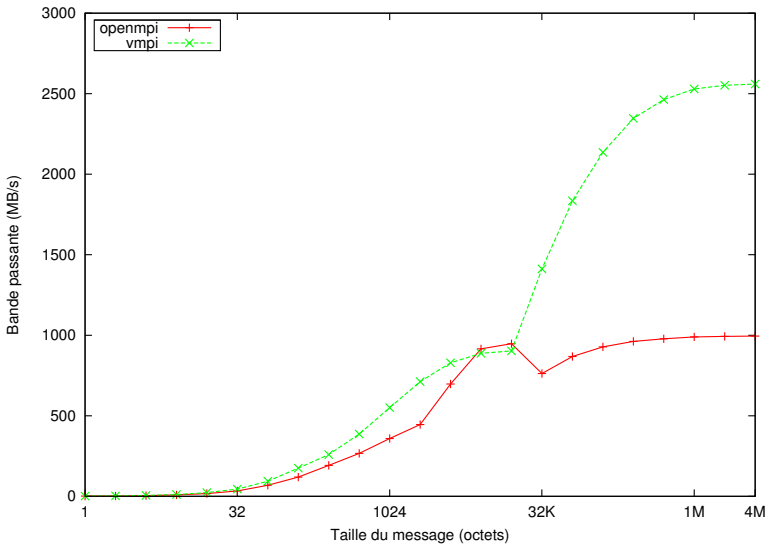
FIGURE 5.3: Latence et bande passante pour le test PingPong sur un noeud (Nehalem)

Ce résultat est attendu, puisque, pour les petits messages, si les mécanismes de partage de mémoire utilisés par OpenMPI et VMPI diffèrent, les transferts de données utilisent ensuite des structures de données similaires. Pour les gros messages en revanche, notre périphérique virtuel permet d'éviter les copies intermédiaires ce que ne propose pas OpenMPI en standard. En effet, effectuer efficacement des copies directes entre processus nécessite l'utilisation d'un module noyau, ce qui est difficile à mettre en place pour une bibliothèque qui se veut portable. On met ici en évidence l'un des avantages de la virtualisation qui permet d'inclure des optimisations dans le noyau des machines virtuelles sans perturber le fonctionnement de l'hôte ou risquer d'introduire des failles de sécurité.

La figure 5.4 présente les résultats du même test PingPong sur un noeud de la machine Fortoy. Nous comparons les résultats obtenus lorsque les 2 tâches MPI sont exécutées directement sur l'hôte, avec la bibliothèque OpenMPI, et dans deux machines virtuelles co-hébergées, avec



(a) Latence



(b) Bande passante

FIGURE 5.4: Latence et bande passante pour le test PingPong sur un noeud (Fortoy)

VMPI. Comme indiqué précédemment, nous ne pouvions pas tester le périphérique paravirtualisé efficace proposé par QEMU sur les noeuds de Fortoy car il nécessite une version du noyau hôte trop récente et un accès root à la machine.

Nous constatons une fois de plus que les performances obtenues pour les petits messages sont similaires dans les cas virtualisés et natifs. En revanche, pour les messages de taille plus importante, la bande passante obtenue par VMPI est bien plus élevée que celle obtenue par OpenMPI. Ceci s'explique toujours par l'utilisation des capacités d'échange de message par copie directe offertes par notre périphérique virtuel. La différence est bien plus marquée que précédemment probablement parce que la bande passante mémoire disponible est plus limitée sur l'architecture Core, qui équipe les noeuds de Fortoy, que sur l'architecture Nehalem. Le coût des copies intermédiaires employées par OpenMPI est donc plus pénalisant dans ce cas.

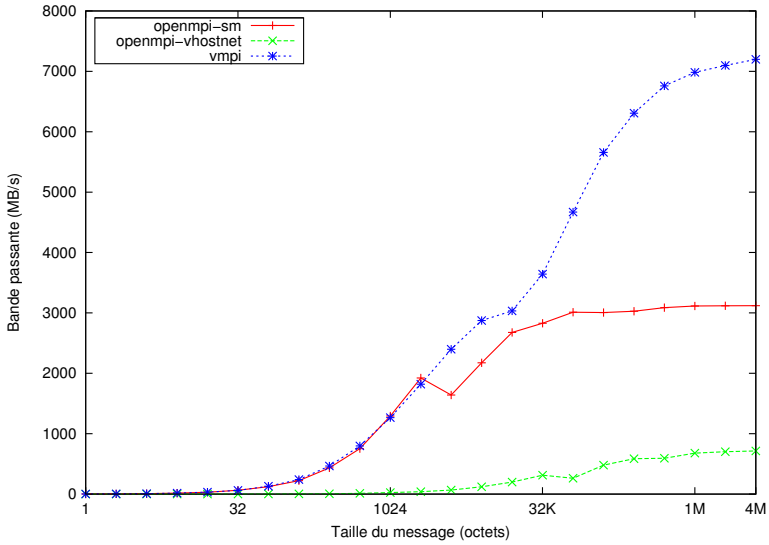


FIGURE 5.5: Bande passante pour le test SendRecv sur un noeud (Nehalem)

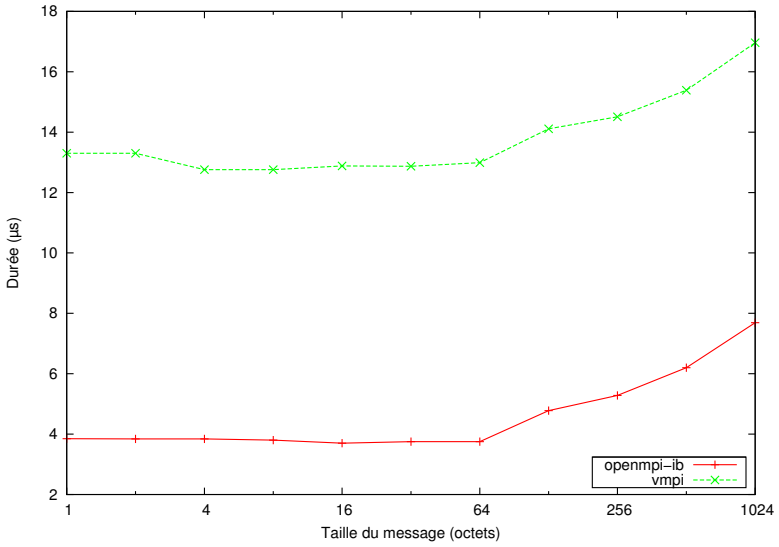
Les résultats du test SendRecv présentés sur la figure 5.5 permettent de confirmer cette théorie. Ce test est similaire au test PingPong à la différence que les deux messages sont échangés simultanément ce qui exige plus de bande passante mémoire. On constate alors que la bande passante supérieure offerte par l'architecture Nehalem ne permet plus de masquer le coût de la copie intermédiaire utilisée par OpenMPI.

Communications inter-noeud

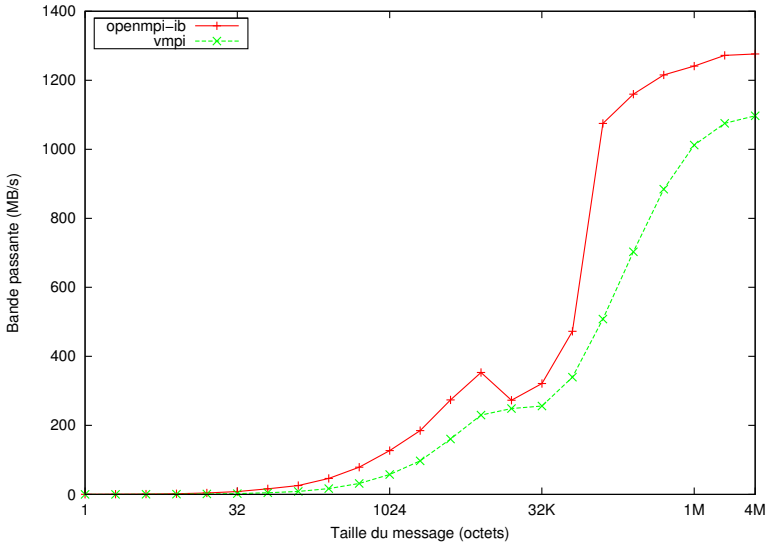
Nous étudions maintenant les performances des communications point-à-point lorsque les tâches MPI sont exécutées sur des noeuds différents. La figure 5.6 présente les résultats du test PingPong sur la machine Fortoy. Nous comparons les performances obtenues lorsque les tâches MPI sont exécutées directement sur les noeuds hôtes avec la bibliothèque OpenMPI, en utilisant le réseau Infiniband natif (courbe intitulée *openmpi-ib*), à celles obtenues avec la bibliothèque VMPI (courbe intitulée *vmpi*). Comme nous l'avons vu précédemment, notre composant en espace utilisateur s'appuie actuellement sur MPI pour communiquer entre les hôtes, et c'est la même bibliothèque OpenMPI qui est utilisée pour cela.

Pour les petits messages on constate que la virtualisation a ici un coût important puisque chaque envoi de message nécessite un changement de contexte entre hôte et invité. Cela induit une latence supplémentaire d'environ 9µs pour le transfert d'un message de 4 octets. Cette latence supplémentaire ne peut être évitée dans le cas général si l'on ne souhaite pas laisser les machines virtuelles contrôler directement les périphériques physiques de communication. Cependant, l'efficacité du support matériel de la virtualisation s'améliorant à chaque nouvelle version de processeur, cette latence est amenée à diminuer. Ceci peut se vérifier sur la figure 5.7 qui présente le résultat du même test exécuté sur les noeuds Jack, équipés de processeurs de dernière génération. La latence supplémentaire induite par la virtualisation tombe à 4µs, bien que ces processeurs soient de fréquence inférieure à ceux qui équipent la grappe Fortoy. Enfin, notons que cette latence reste bien moindre que celle introduite par le périphérique Ethernet virtuel implémenté par QEMU qui dépassait déjà 50µs pour le test en mémoire partagée (voir Figure 5.3).

En ce qui concerne les gros messages, on remarque que la bande passante maximale atteinte

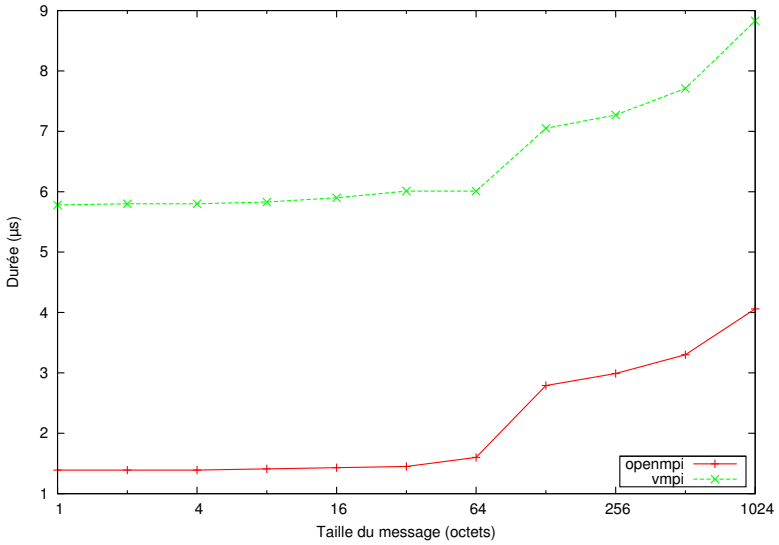


(a) Latence

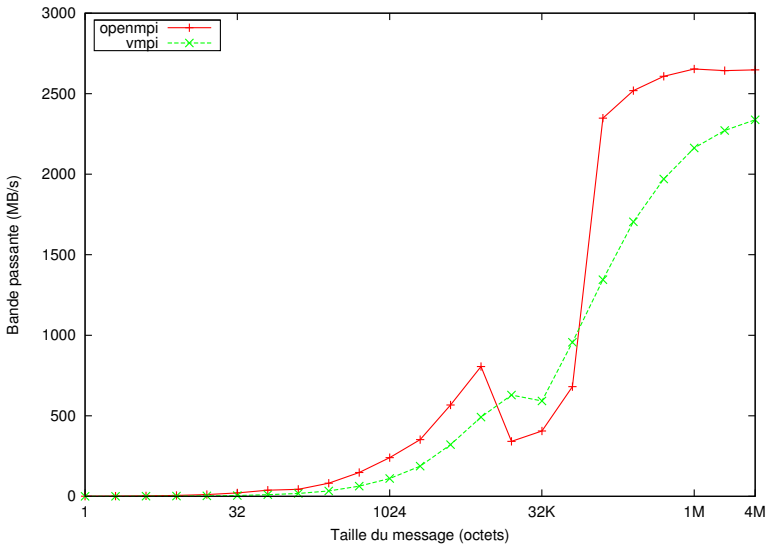


(b) Bande passante

FIGURE 5.6: Latence et bande passante pour le test PingPong entre deux noeuds (Fortoy)



(a) Latence



(b) Bande passante

FIGURE 5.7: Latence et bande passante pour le test PingPong entre deux noeuds (Jack)

par VMPI sur les noeuds de Fortoy et de Jack est inférieure de respectivement 14% et 12% à la bande passante atteinte par OpenMPI sur ces même machines. Ce surcoût vient du fait que notre implémentation actuelle utilise elle-même MPI pour effectuer les transferts réseau ce qui n'est pas optimal. En effet, les messages les plus volumineux utilisent le mécanisme d'accès mémoire distant proposé par notre périphérique virtuel ce qui amène à transférer sur le réseau l'ensemble des pages physiques virtuelles correspondant au message. Ces données n'étant pas contiguës du point de vue du composant en espace utilisateur hôte, cela conduit à effectuer chaque transfert à l'aide d'un type dérivé construit pour l'occasion décrivant la zone mémoire à transmettre. On constate un surcoût similaire pour les messages de taille moyenne. OpenMPI effectue en effet une copie intermédaire pour transférer les tampons de communication de notre périphérique virtuel alors qu'ils pourraient être envoyés directement sur le réseau. Nous prévoyons donc d'implémenter les transferts réseau à l'aide d'une bibliothèque de communication

	8 coeurs (1 noeud)	64 coeurs (8 noeuds)
openmpi	4.4µs	564µs
openmpi-hierarch	-	24.4µs
vmpi	3.1µs	49.5µs

TABLE 5.2: Temps d'exécution d'une barrière en mémoire partagée et sur huit noeuds (Fortoy)

de plus bas-niveau afin de nous affranchir de ces coûts.

5.2.2.2 Communications collectives

Nous évaluons maintenant les performances des communications collectives. Nous utilisons pour cela les tests Bcast, Gather et Barrier, qui illustrent différents schémas de communication.

Communications en mémoire partagée

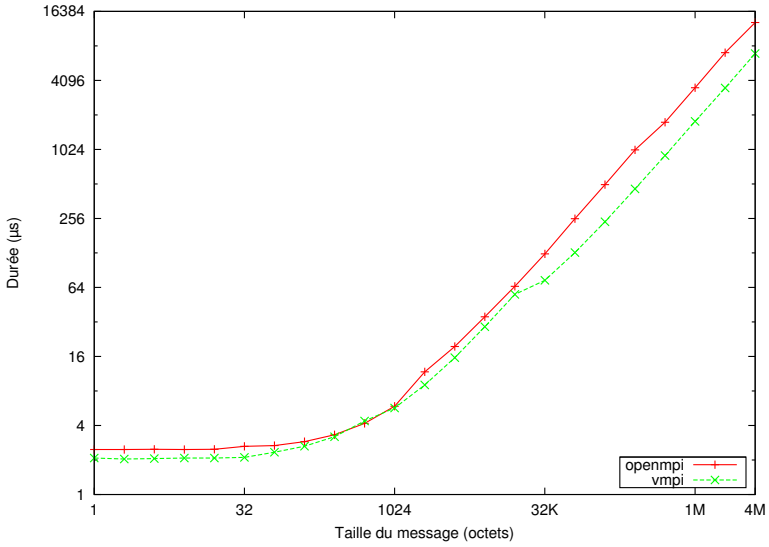
La figure 5.8 présente les résultats obtenus pour les tests Bcast et Gather en mémoire partagée, lorsque 8 tâches MPI sont exécutées sur les 8 coeurs d'un noeud de Fortoy. Nous comparons toujours une exécution native avec la bibliothèque OpenMPI à une exécution virtualisée avec VMPI.

Les résultats sont proches de ceux obtenus lors des tests de communication point-à-point précédents. On constate notamment que, pour les grosses quantités de données, les accès mémoire distants implémentés par notre périphérique virtuel permettent de diviser par près de deux la durée de ces deux opérations collectives par rapport à une exécution native avec OpenMPI. En outre, le léger écart de latence constaté précédemment sur les petits messages est ici amplifiée puisque plusieurs messages successifs sont échangés pour chaque opération collective. Un constat similaire peut être fait pour la durée d'exécution d'une barrière, présentée sur le tableau 5.2.

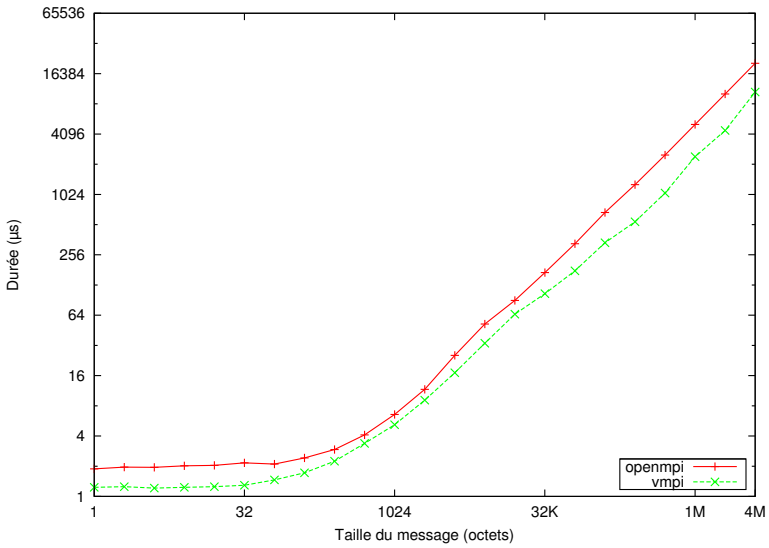
Communications inter-noeud

Nous évaluons maintenant les performances des communications collectives inter-noeud en effectuant les mêmes tests sur 8 noeuds de Fortoy. Les résultats sont présentés sur la figure 5.9. Étonnamment, pour les opérations Barrier et Bcast, les performances d'OpenMPI dans la configuration par défaut sont moins bonnes que celles de VMPI qui se base pourtant sur OpenMPI pour effectuer ses communications inter-noeud. Ceci s'explique par le fait que, par défaut, OpenMPI n'utilise pas de communications collectives hiérarchiques et sature donc le réseau avec un trop grand nombre de messages inter-noeud. On obtient de meilleures performances en forçant OpenMPI à utiliser des communications collectives hiérarchiques (courbes intitulées openmpi-hierarch). Cependant, cette option cause un blocage machine hébergeant la première tâche MPI pour le test Gather, c'est pourquoi la courbe correspondante n'est pas présentée.

Pour les plus gros volumes de données, la bande passante réseau est dans tous les cas saturée par les envois concurrents. De ce fait, l'efficacité moindre de VMPI dans les transferts inter-noeud constatée dans les tests de communication point-à-point est moins pénalisante. En outre, puisque les transferts intra-noeuds sont effectués plus rapidement, le temps total de l'opération est environ 10% plus faible avec VMPI pour une diffusion de 4MiB.

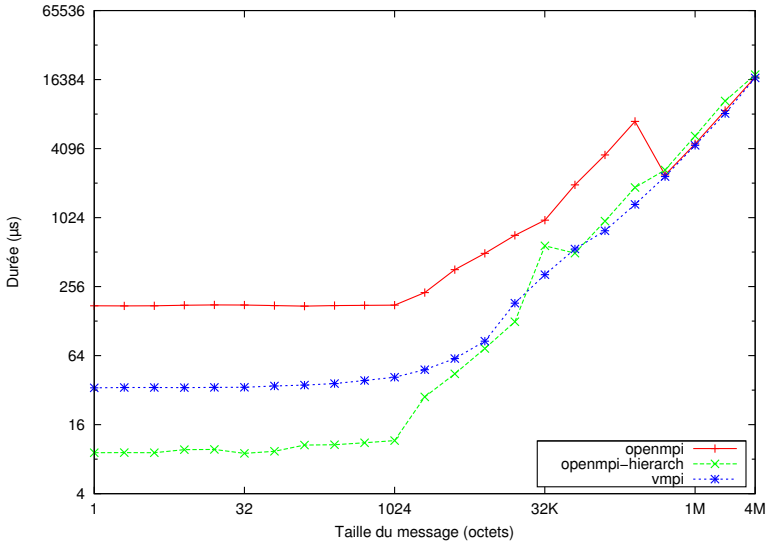


(a) MPI_Bcast

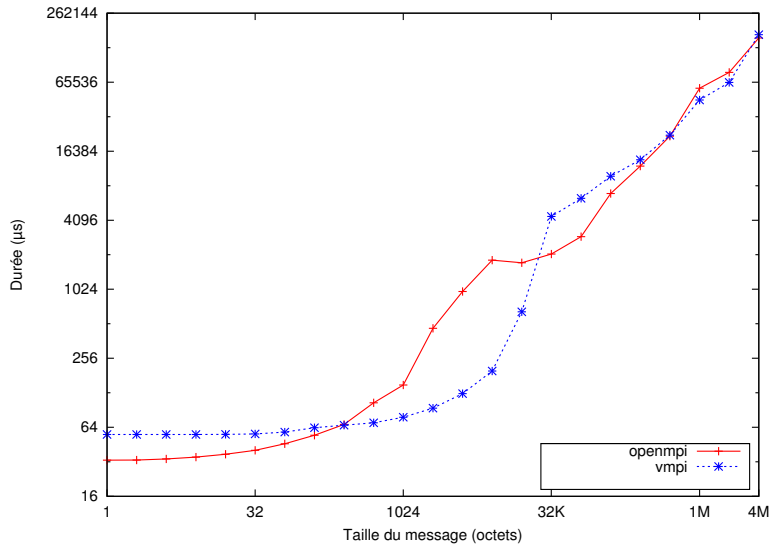


(b) MPI_Gather

FIGURE 5.8: Latence des opérations MPI_Bcast et et MPI_Gather sur un noeud (Fortoy)



(a) MPI_Bcast



(b) MPI_Gather

FIGURE 5.9: Latence des opérations MPI_Bcast et et MPI_Gather sur huit noeuds (Fortoy)

5.3 Évaluation sur des logiciels de calcul

Après avoir étudié les performances brutes des communications en mode natif et virtualisé, nous souhaitons déterminer l'impact des différences de performances constatées sur le temps d'exécution de codes de calcul scientifique typiques. Nous présentons donc, dans cette section, le résultat de deux tests de performances standards, le LINPACK et la suite NAS Parallel Benchmarks. Nous nous intéressons ensuite aux performances de la plateforme de calcul HERA développée au CEA.

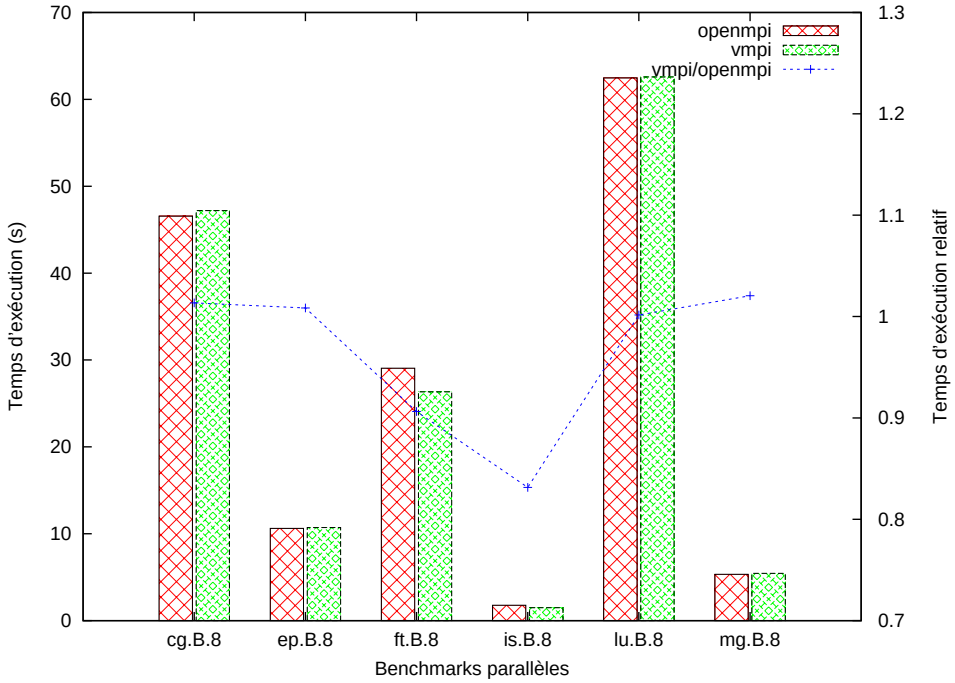


FIGURE 5.10: Temps d'exécution des NAS sur un noeud

5.3.1 NAS Parallel Benchmarks

Les NAS Parallel Benchmarks (NPB) sont une suite de tests de performance écrits par la NASA. Ils consistent en un ensemble de noyaux de calcul et de pseudo-applications parallèles représentatifs des codes de simulation numérique développés pour la mécanique des fluides. Ils sont fréquemment utilisés dans la littérature scientifique pour évaluer les performances des diverses implémentations de MPI.

La figure 5.10 présente les temps d'exécution obtenus lorsque l'on exécute ces tests sur 8 coeurs d'un noeud de Fortoy. Nous comparons toujours une exécution native de 8 tâches MPI utilisant la bibliothèque OpenMPI à une exécution virtualisée où les 8 tâches MPI sont exécutées dans 8 machines virtuelles co-hébergées et utilisent VMPI. On présente les résultats de tous les tests à l'exception de SP et BT qui ne peuvent s'exécuter qu'avec un nombre carré de tâches. Les résultats obtenus dans les cas natifs et virtualisés sont similaires pour la plupart des benchmarks et les différences de temps d'exécution sont inférieures à 2%, mis à part pour les benchmarks FT et IS, où l'exécution avec VMPI est plus rapide de respectivement et 10 et 17%. Ceci s'explique par le fait que ces deux benchmarks sont ceux qui utilisent le plus de messages de grande taille. Ils tirent donc le plus partie de la bande passante supplémentaire dégagée par les copies directes réalisées par VMPI.

La figure 5.11 compare à nouveau les résultats obtenus par OpenMPI et VMPI pour cette suite de test, mais elle est cette fois-ci exécutée sur 8 noeuds de Fortoy, soit 64 coeurs. Dans cette configuration, plusieurs tests montrent un surcoût sensible avec VMPI, allant notamment jusqu'à 40% pour le test IS. Ceci s'explique comme précédemment par le fait que ce test est très

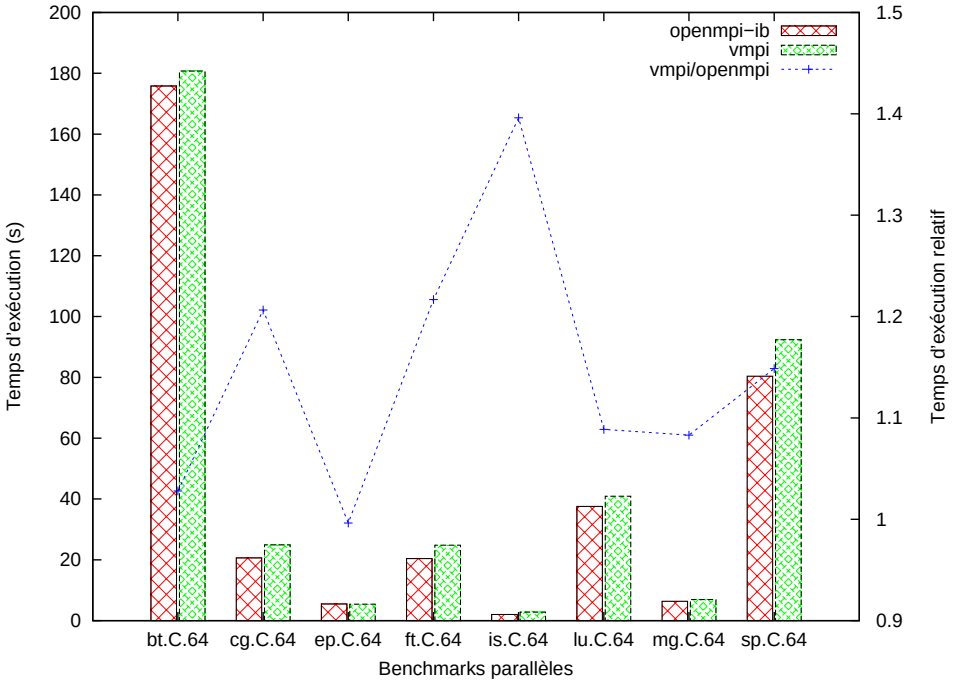


FIGURE 5.11: Temps d'exécution des NAS sur une grappe de 64 coeurs

sensible à la bande passante entre les tâches MPI. La perte de bande passante inter-noeud constatée précédemment est donc ici très pénalisante. À titre de comparaison, l'exécution native du même test en utilisant OpenMPI avec IP-over-Infiniband à la place d'Infiniband natif conduit à un surcoût de plus de 1000%.

Une autre source de perte de performance, qui n'a pas été mise en évidence par les micro-évaluations précédentes vient des communications inter-noeud concurrentes déclenchées sur chaque noeud. En effet, notre implémentation actuelle se base sur MPI pour effectuer les différentes communications inter-noeud. Or, les implémentations MPI existantes supportent mal le mode MPI_THREAD_MULTIPLE qui permet d'effectuer des communications concurrentes depuis plusieurs threads d'un même processus. Nous avons donc dû placer des verrous autour des divers appels à MPI ce qui empêche plusieurs machines virtuelles hébergées sur un même noeud de communiquer simultanément avec des machines virtuelles hébergées sur d'autres noeud. Ce problème n'était pas apparent lors des tests de communications collectives car le mécanisme de communications collectives hiérarchiques permet de limiter cette situation.

Ce résultat appelle plusieurs observations. D'une part cela confirme la nécessité d'utiliser une bibliothèque de communication plus bas-niveau pour implémenter les communications inter-noeud de notre composant en espace utilisateur. D'autre part, cela montre qu'il pourrait être intéressant de migrer les machines virtuelles qui communiquent entre elles sur le même noeud afin de favoriser les communications en mémoire partagée. L'influence du placement des tâches MPI sur les performances à par exemple été étudié dans [81].

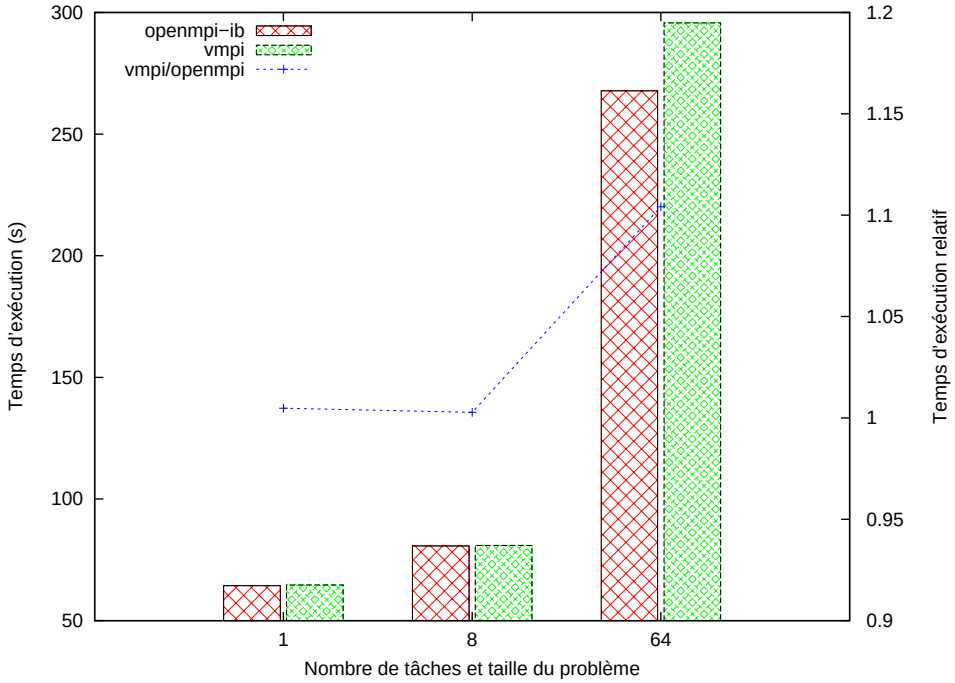


FIGURE 5.12: Temps d'exécution du LINPACK

5.3.2 High Performance LINPACK

Le High Performance LINPACK (HPL) est un test qui mesure le temps nécessaire à résolution d'un système linéaire dense en utilisant une factorisation LU. Ce test est particulièrement important dans la communauté du calcul haute performance, puisque c'est celui qui est utilisé pour classer les machines au sein du Top500. Les algorithmes de résolution de systèmes linéaires dense permettent en effet de très bien exploiter les capacités de calcul flottant des processeurs modernes. En particulier, ils peuvent être implémentés à l'aide des BLAS (pour Basic Linear Algebra Subprograms) qui sont un ensemble d'opérations couramment utilisées en algèbre linéaire, telles que les multiplications de matrices. Des bibliothèques de BLAS, telles que ATLAS, Goto BLAS, ou encore la MKL d'Intel fournissent des implémentations très optimisées de ces opérations pour différents processeurs. On s'approche ainsi de la puissance de calcul théorique des machines en terme de nombre d'opérations flottantes effectuées par seconde.

La figure 5.12 présente les temps d'exécution comparés en mode natif avec OpenMPI et virtualisé avec VMPI, sur Fortoy. On fait varier le nombre de tâches MPI utilisées de 1 à 64 et on utilise toujours autant de coeurs physiques que de tâches. Les exécutions à 1 et 8 tâches sont donc effectuées sur un seul noeud, tandis que l'exécution à 64 tâches est effectuée sur 8 noeuds.

Le résultat de l'exécution séquentielle nous permet de vérifier que le surcoût lié à la virtualisation du processeur reste faible, même pour des codes très optimisés tels que le LINPACK. Cependant, pour arriver à ce résultat, il nous a fallu prendre soin d'exposer aux machines virtuelles des processeurs virtuels ayant les mêmes identifiants CPUID que le processeur hôte. En effet la bibliothèque de BLAS MKL, que nous avons utilisé dans ce test, se base sur cet identi-

fiant pour choisir des routines spécialement optimisées pour chaque version de processeur. Ce type de comportement nécessite une prise en charge particulière dans le cadre de la virtualisation, puisque cela peut empêcher de migrer les machines virtuelles entre des machines ayant des processeurs de différents types. Il faut dans ce cas déterminer un identifiant de processeur correspondant à un « plus petit dénominateur commun » des fonctionnalités supportées par les processeurs hôtes potentiels.

Le résultat des exécutions parallèles montre, comme dans le cas des NPB, qu'il n'y a aucun surcoût pour l'exécution de tâches parallèles en mémoire partagée. Pour une exécution sur 8 noeuds, la perte de performance induite par la virtualisation est de l'ordre de 10% ce qui reste raisonnable.

5.3.3 Plateforme AMR d'hydrodynamique : HERA

Depuis quelques années, des études sont engagées au CEA/DAM sur des méthodes et algorithmes de calcul utilisant des maillages AMR (Adaptive Mesh Refinement). HERA (pour Hydrodynamique Euler Raffinement Adaptatif) est une plate-forme multi-physique AMR développée au CEA/DAM [82]. Ce logiciel permet de simuler des écoulements compressibles multifluides (domaine de la Dynamique Rapide) en plusieurs dimensions d'espace, avec couplage à des modèles physiques variés, comme des phénomènes thermiques de diffusion non linéaires. HERA est composé principalement de 300 000 lignes de C++, mais fait également appel à des langages interprétés comme Python ou C#. La méthode de parallélisation choisie est l'approche SPMD par décomposition de domaines. La parallélisation des schémas numériques par cette approche est classique : aux points de synchronisation des algorithmes, pour prendre en compte les contributions des sous-domaines voisins, des échanges de messages s'opèrent pour mettre à jour les bords de chaque sous-domaine. En revanche, l'approche AMR couplée à la décomposition de domaine induit des déséquilibrages de charge entre les tâches parallèles. Le maillage s'adaptant au cours de la simulation, les charges de travail attribuées à chaque tâche évoluent au cours du calcul. Des schémas de communications très agressifs sont donc mis en place pour redistribuer périodiquement les données entre les tâches.

La figure 5.13 présente les temps d'exécution de HERA pour un cas-test en trois dimensions simulant l'évolution hydrodynamique d'une coquille sphérique décrite par une équation d'état stiffened gas et d'un gaz décrit par l'équation d'état gaz parfait. Nous utilisons deux maillages, composés respectivement d'un million de mailles (1 octant) et de huit millions de mailles (8 octants), chaque maillage disposant de 3 niveaux de raffinement possibles. Comme pour les tests précédents, on compare le mode d'exécution natif avec OpenMPI au mode d'exécution virtualisé avec VMPI sur la machine Fortoy. On étudie le cas où le code est exécuté de manière séquentielle ainsi que celui où il est parallélisé sur tous les coeurs d'un noeud puis de 8 noeuds. Ce test permet de valider l'utilisation de la virtualisation, y compris pour l'exécution de véritables simulations numériques puisque le surcoût en performance reste dans tous les cas inférieur à 5%.

5.4 Migrations de machines virtuelles

Notre support exécutif permet de faire profiter simplement n'importe quelle application parallèle MPI des bénéfices apportés par la migration. En particulier, il devient possible, de façon transparente, de migrer d'un hôte à l'autre les différentes tâches d'une application MPI au cours

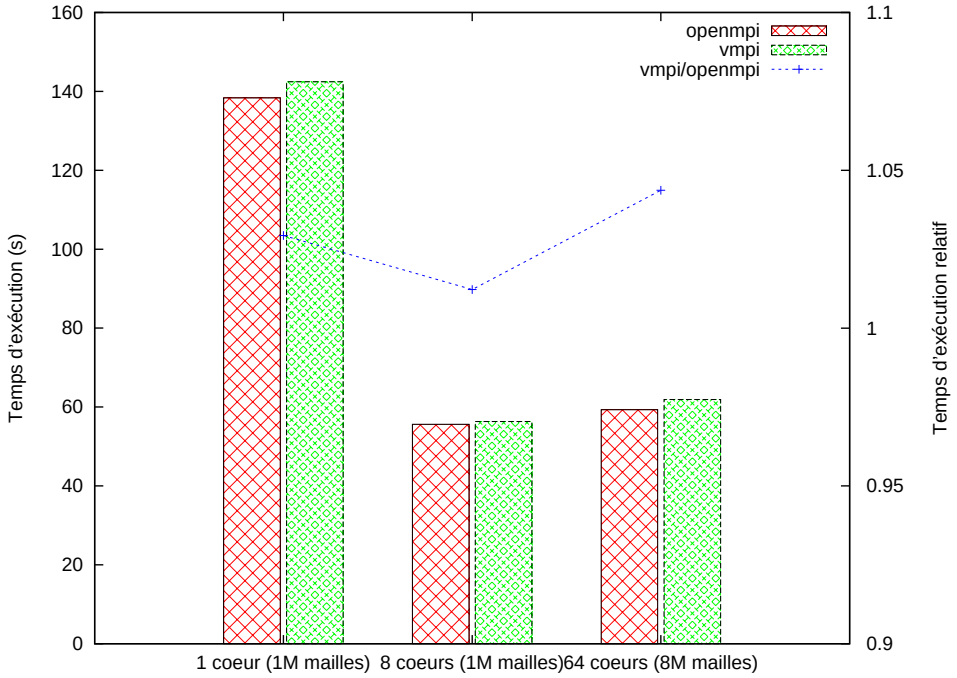


FIGURE 5.13: Temps d'exécution de HERA

de son exécution. Dans cette section, nous présentons les résultats de différents tests évaluant les performances de cette opération.

5.4.1 Performances des communications pendant les migrations

Notre premier test élémentaire met en évidence l'influence des migrations sur les performances des communications entre machines virtuelles. Pour cela, nous effectuons en boucle mille échanges de messages de taille nulle entre deux tâches MPI et mesurons la latence moyenne constatée à chaque itération de la boucle. Ces deux tâches sont exécutées dans deux machines virtuelles exécutées initialement sur deux noeuds de Fortoy. Toutes les dix secondes une migration est déclenchée afin de faire passer la deuxième tâche d'un hôte à l'autre. Les résultats sont présentés sur la figure 5.14.

A l'issue de la première migration, les machines virtuelles sont hébergées sur le même hôte ce qui permet aux tâches MPI de communiquer directement en mémoire partagée. On constate donc une diminution de la latence dès la première itération de la boucle suivant la migration, ce qui montre que le flot de messages échangés n'est pas interrompu longtemps. À l'inverse, à la vingtième seconde, la deuxième machine virtuelle retourne sur son hôte d'origine. Le coût de la migration est reflété sur la première itération de la boucle, puis on retrouve la latence initiale des communications inter-noeud à partir des itérations suivantes.

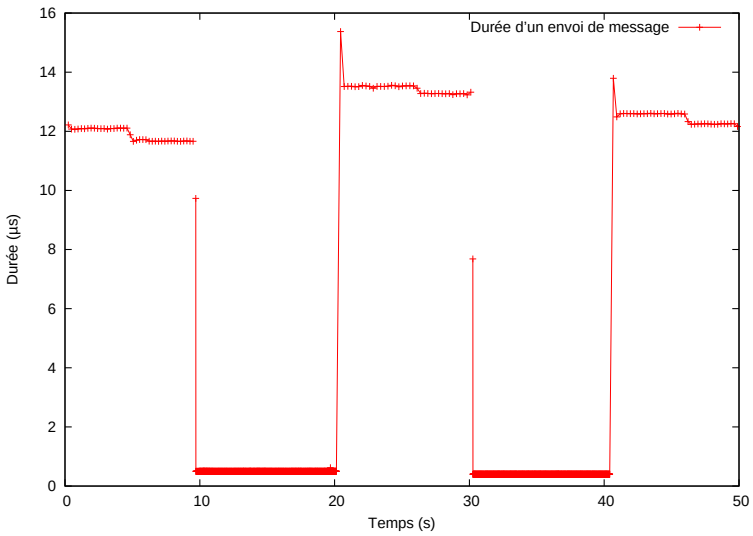


FIGURE 5.14: Performances des communications pendant les migrations

5.4.2 Impact sur le temps d'exécution de HERA

Nous nous intéressons maintenant à l'impact des migrations sur les performances d'une application parallèle. Nous utilisons pour cela la plateforme HERA et nous mesurons le temps d'exécution du cas-test décrit section 5.3.3, lorsque des migrations ont lieu. Nous étudions différentes configurations et présentons, pour chacune d'elle, le différentiel de temps par rapport à une exécution sans migration. Nous mesurons en outre le temps écoulé entre le début et la fin de l'ensemble des migrations, ainsi que le volume total de données transférées. Tous les tests sont exécutés sur la grappe Fortoy et les résultats sont reportés sur la figure 5.15.

Détail des configurations testées

– 1/8 : Pour ce premier test, nous utilisons 8 tâches MPI virtualisées pour exécuter HERA avec le cas-test à un million de mailles. Initialement, les 8 tâches MPI sont exécutées sur 8 noeuds de Fortoy. Après dix secondes d'exécution, nous migrons une des machines virtuelles sur un neuvième noeud. Les conditions d'exécution sont donc les mêmes avant et après la migration et le différentiel de temps d'exécution relevé – 1.4 seconde – est intégralement lié à la migration. L'impact d'une migration sur le temps d'exécution d'une application parallèle est donc faible.
– 8/64 : Pour le test suivant HERA est exécuté dans 64 tâches MPI virtualisées et traite le cas-test à huit millions de mailles. Les machines virtuelles sont toujours exécutées sur huit noeuds de Fortoy. On teste cette fois-ci le coût de migrations simultanées, puisque l'on fait migrer simultanément les huit machines virtuelles d'un des noeud vers un neuvième noeud. Une fois encore, les conditions d'exécution sont les même avant et après la migration ce qui permet de relever uniquement l'impact de la migration sur le temps d'exécution. Celui-ci est de 6.1 secondes, la quantité de données à transférer étant de 2.1GiB, soit près de 10 fois plus que dans le cas précédent. Ce temps reste faible devant le temps d'exécution typique d'applications de calcul intensif.

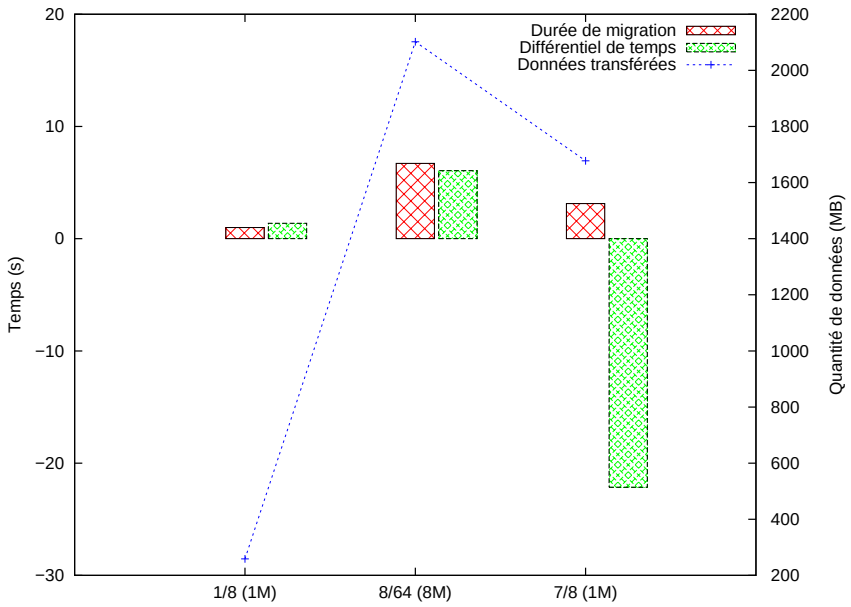


FIGURE 5.15: Impact des migrations sur le temps d'exécution de HERA

– 7/8 : Ce dernier test a pour but de mettre en évidence l'intérêt des migrations de machines virtuelles pour exploiter des ressources de calcul inutilisées. On exécute HERA dans 8 tâches MPI virtualisées pour traiter le cas-test à un million de mailles. Initialement les 8 machines virtuelles sont co-hébergées sur un noeud de Fortoy. On a donc une machine virtuelle par coeur. Au bout de dix secondes, sept machines virtuelles sont migrées, afin que chaque machine virtuelle soit hébergée sur son propre noeud. Bien que la durée de migration soit de 3.1 secondes, le temps total d'exécution est réduit de 22 secondes par rapport au cas où les machines virtuelles restent sur le noeud initial, soit une reduction d'environ un tiers. En effet, chaque tâche MPI dispose alors d'une bande passante mémoire bien supérieure, ainsi que de la totalité de la mémoire cache de son processeur.

5.5 Bilan des évaluations

Dans ce chapitre, nous avons tout d'abord mesuré les performances brutes de notre environnement d'exécution à l'aide de différents tests élémentaires. Cela nous a permis de vérifier qu'il est possible d'instancier une grappe de machines virtuelles pour exécuter une application parallèle, et ce sans incidence significative sur le temps de lancement de l'application. Nous avons par ailleurs constaté les très bonnes performances des opérations MPI en mémoire partagée grâce aux accès mémoire distants virtuels implémentés par notre périphérique. En revanche, les communications inter-noeud affichent une perte d'efficacité par rapport aux communications natives. En sus de la latence liée aux basculements entre hôte et invité, cette perte d'efficacité vient du fait que nous utilisons MPI pour implémenter les communications entre les noeuds.

Nous avons ensuite étudié l'impact de ces variations de performances sur l'exécution de lo-

giciels représentatifs des applications parallèles de calcul scientifique. Les tests NAS nous ont confirmé que, pour certaines applications très sensibles aux performances des communications, il nous faudra améliorer notre implémentation en nous affranchissant de MPI pour nous rapprocher des performances natives. Cependant nos autres tests applicatifs montrent que notre solution permet déjà d'exécuter des calculs parallèles dans des machines virtuelles avec un surcoût en temps d'exécution inférieur à 10%. En particulier, nos tests sur la plateforme HERA ont permis de confirmer la robustesse de notre solution ainsi que sa capacité à exécuter efficacement de véritables smulations numériques tout en bénéficiant des avantages apportés par la virtualisation tels que la possibilité de migrer les machines virtuelles.

Conclusion et perspectives

Contexte de l'étude

L'augmentation continuelle de la demande en puissance de calcul a conduit au développement d'architectures matérielles de plus en plus parallèles et donc de plus en plus difficiles à programmer, à utiliser, et à administrer. De nouvelles solutions doivent être envisagées afin de minimiser l'impact de ces évolutions matérielles. En particulier la virtualisation, qui a connu un regain d'intérêt important ces dernières années, semble apporter des réponses intéressantes à ces problèmes. Toutefois, pour que cette technologie soit acceptée dans le cadre du calcul intensif son impact sur les performances doit être faible, et elle doit pouvoir être mise en oeuvre de la manière la plus transparente possible pour les utilisateurs.

Les travaux réalisés dans cette thèse ont été menés conjointement au laboratoire PNP (Pôle Numérique et Prospectives) au CEA/DAM Île de France et au sein du projet INRIA RUNTIME. Le laboratoire PNP a pour objectif l'élaboration de codes de calculs scientifiques avant-gardistes et hautes performances tandis que l'équipe RUNTIME a pour objectif l'étude, la conception et la réalisation de supports exécutifs performants pour architecture parallèle.

Contributions

En étudiant l'état de l'art des techniques de virtualisation, ainsi que l'architecture matérielle des grappes de calcul modernes, nous avons identifié la principale source de surcoût en performance comme étant la virtualisation des périphériques réseau haute performances. C'est en effet grâce à l'efficacité des réseaux rapides que les codes de calcul peuvent être parallélisés efficacement à grande échelle en utilisant notamment la programmation par passage de message. Nous avons donc cherché à améliorer les performances des communications entre machines virtuelles, que celles-ci soient co-hébergées ou hébergées sur différents noeuds d'une grappe.

Pour cela, nous avons conçu un périphérique virtuel de communication dont l'interface est bien adaptée au passage de messages en contexte virtualisé. En effet, cette interface a été étudiée de manière à ce que les échanges de messages puissent tirer tout aussi efficacement parti des fonctionnalités des périphériques réseau rapides que des caractéristiques des architectures à mémoire partagée. Une bibliothèque MPI exploitant ce périphérique a ensuite été développée afin que des applications parallèles existantes puissent bénéficier des performances de notre périphérique virtuel.

Par ailleurs, notre objectif était de permettre à des utilisateurs d'une grappe de bénéficier des avantages apportés par la virtualisation de la manière la plus transparente possible. Dans cette optique, nous proposons un environnement d'exécution permettant de déployer une application parallèle dans une grappe de machines virtuelles aussi simplement et aussi rapidement

que si cette application était exécutée directement sur la grappe hôte. Cet environnement se base notamment sur une opération de fork pour machines virtuelles permettant de répliquer très rapidement une machine virtuelle autant de fois que nécessaire pour exécuter une application parallèle donnée.

Notre solution a été implémentée avec le souci de s'intégrer facilement à une grappe existante basée sur Linux. Ainsi, nous nous sommes basés sur l'hyperviseur KVM qui est intégré en standard au noyau Linux, et notre code a pu être entièrement écrit en espace utilisateur. N'importe quel utilisateur d'une grappe peut alors facilement déployer des applications dans des machines virtuelles sans affecter le fonctionnement des programmes natifs. En outre, les mécanismes de console virtuelle ainsi que de partage de système de fichiers que nous avons implémentés facilitent les interactions entre codes exécutés de manière native et virtualisée.

Nous avons évalué les performances de notre solution en comparant les temps d'exécution obtenus pour différentes applications parallèles lorsqu'elles sont exécutées nativement ou dans une grappe de machines virtuelles. Nous obtenons d'excellents résultats lorsque les machines virtuelles sont co-hébergées, puisque les performances obtenues avec notre bibliothèque MPI sont supérieures à celles obtenues nativement avec la bibliothèque standard OpenMPI. Ce résultat met en évidence un des avantages de la virtualisation qui est de permettre d'inclure des optimisations dans le noyau des machines virtuelles sans causer de problème de portabilité ou de sécurité. Lorsque les machines virtuelles sont réparties sur plusieurs noeuds, on constate un surcoût en performance qui reste inférieur à 10% pour la plupart des applications testées. Ce résultat devrait s'améliorer lorsque nous aurons implémenté les communications entre les noeuds à l'aide d'une bibliothèque de communication bas-niveau. Enfin, nous avons évalué notre environnement d'exécution sur une application de production totalisant plus de 300 000 lignes de code baptisée HERA. Ce logiciel de simulation numérique emploie des maillages adaptatifs et fait une utilisation très agressive des communications, notamment pour maintenir l'équilibrage de charge entre les tâches. Cela nous a donc permis de valider la robustesse de notre implémentation, ainsi que son efficacité sur une application de cette ampleur.

Perspectives

De nombreuses pistes existent pour poursuivre ces travaux. Tout d'abord, plusieurs idées restent à explorer pour améliorer l'implémentation de notre support exécutif. En outre, les fonctionnalités que nous avons proposé dans cette thèse ouvrent la voie à de nouvelles expérimentations concernant l'utilisation de la virtualisation dans le cadre du calcul haute performance.

Évolutions de l'implémentation

Dans un futur proche, nous souhaitons améliorer l'implémentation des transferts de données inter-noeud effectués dans le cadre de l'émulation de notre périphérique virtuel de communication. En effet, les évaluations que nous avons conduit ont permis de montrer les limites de l'implémentation actuelle basée sur MPI. Pour les petits messages, les bibliothèques MPI effectuent des copies intermédiaires qui sont superflues puisque nous utilisons toujours les mêmes tampons de communications qui pourraient être directement enregistrés auprès des périphériques réseaux. Pour les gros messages, la nécessité d'utiliser des types dérivés est pénalisante. Enfin, la mauvaise gestion du mode MPI_THREAD_MULTIPLE nous oblige à sérialiser les communications entre les machines virtuelles d'un noeud et le reste de la grappe virtuelle.

Nous comptons donc implémenter les communications inter-noeud en nous basant sur diverses bibliothèques bas-niveau telles que les bibliothèques de *verbs* pour Infiniband. Notre implémentation basée sur MPI sera néanmoins conservée pour la portabilité qu'elle apporte.

Par ailleurs, l'interface bas-niveau de notre périphérique virtuel est proche de l'interface MX utilisée par les réseaux Myrinet. Il serait intéressant d'implémenter une bibliothèque proposant une compatibilité complète avec cette interface. Cela permettrait d'utiliser des bibliothèques MPI existantes supportant le réseau Myrinet pour piloter notre périphérique. Plus généralement, d'autres programmes utilisant directement l'interface MX, tels que certains systèmes de fichiers distribués, pourraient être utilisées efficacement dans des machines virtuelles.

Enfin, nous nous sommes principalement intéressés à l'utilisation de la virtualisation dans le cadre d'une grappe existante afin de maximiser l'applicabilité de notre solution. Nous l'avons donc implémentée au-dessus de l'hyperviseur KVM et nous nous sommes attachés à ce que l'ensemble de notre code soit exécuté en espace utilisateur hôte. Nous souhaitons toutefois étudier les possibilités offertes par l'implémentation de quelques fonctionnalités dans le noyau hôte. Cela permettrait en effet de limiter les latences liées aux changements de contextes entre hôte et invité, puisqu'il est moins coûteux de basculer en espace noyau hôte qu'en espace utilisateur hôte. Dans le même ordre d'idées, l'utilisation d'un hyperviseur de type I permettrait de s'affranchir des problèmes liés au bruit système rencontrés actuellement par les systèmes d'exploitation généralistes tel que Linux.

Exploitation des fonctionnalités offertes par le support exécutif

Nous avons évoqué, dans la section 2.3, de nombreux travaux visant à mettre à profit la possibilité de migrer des machines virtuelles pour améliorer le fonctionnement des ordonnanceurs de travaux utilisés dans la grappe. Il serait donc intéressant d'étudier comment un support exécutif tel que celui que nous avons développé pourrait être utilisé de concert avec un ordonnanceur de travaux de manière à optimiser l'allocation des noeuds d'une grappe.

En particulier, une piste à explorer consisterait à collecter des statistiques d'utilisation du périphérique virtuel afin d'évaluer l'intensité des flux de communication entre les machines virtuelles. On pourrait alors recalculer la distribution des machines virtuelles sur les hôtes, afin que les machines virtuelles qui communiquent le plus entre elles soient hébergées sur un même noeud, ou sur des noeuds proches dans le cas - de plus en plus fréquent avec l'augmentation de la taille des grappes - où tous les noeuds ne sont pas connectés aussi efficacement entre eux. Il a en effet été montré que le placement des tâches MPI en fonction de la topologie réseau sous-jacente joue un rôle important dans les performances obtenues [83].

Par ailleurs, nous souhaitons étudier l'utilisation de systèmes d'exploitation minimaux conçus spécifiquement pour le calcul intensif. En effet nous avons jusqu'à présent évalué notre support exécutif avec des machines virtuelles embarquant un noyau Linux. Néanmoins, l'un des bénéfices de la virtualisation est de donner à chacun la possibilité de modifier le code privilégié qui s'exécute au sein de sa machine virtuelle. Nous avons déjà vu l'intérêt de cette approche puisqu'elle nous a permis d'implémenter des communications en mémoire partagée par copie directe sans avoir à modifier le noyau hôte. Plus généralement, les environnements d'exécution existants dédiés au calcul haute performance doivent souvent déployer des astuces ingénieuses pour contourner les politiques d'allocation mémoire, d'ordonnancement, ou encore de gestion des entrées/sorties mises en place par les systèmes d'exploitation généralistes. La virtualisation offre la possibilité d'intégrer les optimisations nécessaires directement au sein d'un système d'exploiation dédié à l'exécution d'une application parallèle.

Bibliographie

[1] T. STERLING, D. J. BECKER, D. SAVARESE, J. E. DORBAND, U. A. RANAWAKE et C. V. PACKER. Beowulf : A Parallel Workstation For Scientific Computation. Dans *Proceedings of the 24th International Conference on Parallel Processing*, pp. 11–14. CRC Press, 1995.

[2] TOP500. `www.top500.org`.

[3] G. E. MOORE. Cramming More Components Onto Integrated Circuits. *Electronics*, volume 38(8) :pp. 114–117, Avril 1965.

[4] N. J. BODEN, D. COHEN, R. E. FELDERMAN, A. E. KULAWIK, C. L. SEITZ, J. N. SEIZOVIC et W.-K. SU. Myrinet : A Gigabit-per-Second Local Area Network. *IEEE Micro*, volume 15 :pp. 29–36, 1995. ISSN 0272-1732. doi :http://doi.ieeecomputersociety.org/10.1109/40.342015.

[5] INFINIBAND TRADE ASSOCIATION. InfiniBand Architecture Specification : Release 1.0, 2000.

[6] R. RECIO, P. CULLEY, D. GARCIA, J. HILLAND et B. METZLER. An RDMA protocol specification, 2005. `http://www.ietf.org/internet-drafts/draft-ietf-rddp-rdmap-04.txt`.

[7] H. SHAH, J. PINKERTON, R. RECIO et P. CULLEY. Direct data placement over reliable transports, 2005. `http://www.ietf.org/internet-drafts/draft-ietf-rddp-ddp-04.txt`.

[8] G. CHIOLA et G. CIACCIO. GAMMA : A low-cost network of workstations based on active messages. Dans *Fifth Euromicro Workshop on Parallel and Distributed Processing (PDP '97)*, pp. 78–83, 1997.

[9] B. GOGLIN. Design and Implementation of Open-MX : High-Performance Message Passing over generic Ethernet hardware. Dans *CAC 2008 : Workshop on Communication Architecture for Clusters, held in conjunction with IPDPS 2008*. IEEE Computer Society Press, Miami, FL, 2008 2008.

[10] H. JOURDREN et D. DUREAU. Parallélisme et équilibrage de charge en hydrodynamique AMR. *Revue Chocs*, volume 28 :pp. 51–60, October 2003.

[11] PVM : Parallel Virtual Machine. `http://www.csm.ornl.gov/pvm/`.

[12] MESSAGE PASSING INTERFACE FORUM. MPI : A Message-Passing Interface Standard, 1994.

[13] R. L. GRAHAM, T. S. WOODALL et J. M. SQUYRES. Open MPI : A Flexible High Performance MPI. Dans *Proceedings of the 6th Annual International Conference on Parallel Processing and Applied Mathematics*. Poznan, Poland, September 2005.

[14] MPICH2 : High-performance and Widely Portable MPI. `www.mcs.anl.gov/mpi/mpich/`.

[15] W. HUANG, G. SANTHANARAMAN, H.-W. JIN, Q. GAO et D. K. x. D. K. PANDA. Design of High Performance MVAPICH2 : MPI2 over InfiniBand. Dans *CCGRID '06 : Proceedings*

of the Sixth IEEE International Symposium on Cluster Computing and the Grid, pp. 43–48. IEEE Computer Society, Washington, DC, USA, 2006. ISBN 0-7695-2585-7. doi :http://dx.doi.org/10.1109/CCGRID.2006.32.

[16] B. MELTZ. Simulation numérique téraflopique en hydrodynamique. *Revue Chocs*, volume 28 :pp. 25–32, 2003.

[17] R. LOTTIAUX, P. GALLARD, G. VALLEE, C. MORIN et B. BOISSINOT. OpenMosix, OpenSSI and Kerrighed : a comparative study. Dans *CCGRID '05 : Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05) - Volume 2*, pp. 1016–1023. IEEE Computer Society, Washington, DC, USA, 2005. ISBN 0-7803-9074-1.

[18] M. FRIGO, C. E. LEISERSON et K. H. RANDALL. The Implementation of the Cilk-5 Multithreaded Language. Dans *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pp. 212–223. Montreal, Quebec, Canada, June 1998.

[19] Intel Threading Building Blocks. `http://www.threadingbuildingblocks.org/`.

[20] C. H. KOELBEL, D. B. LOVEMAN, R. S. SCHREIBER, G. L. STEELE, Jr. et M. E. ZOSEL. *The high performance Fortran handbook*. MIT Press, Cambridge, MA, USA, 1994. ISBN 0-262-61094-9.

[21] OPENMP FORUM. OpenMP. `http://www.openmp.org`.

[22] M. PÉRACHE, H. JOURDREN et R. NAMYST. MPC : A Unified Parallel Runtime for Clusters of NUMA Machines. Dans *Proceedings of the 14th International Euro-Par Conference*, volume 5168 of *Lecture Notes in Computer Science*, pp. 78–88. Springer, Las Palmas de Gran Canaria, Spain, August 2008. ISBN 978-3-540-85450-0. doi :10.1007/978-3-540-85451-7_9. `http://runtime.bordeaux.inria.fr/Download/Publis/PerJouNam08EuroPar.pdf`.

[23] F. TRAHAY, É. BRUNET et A. DENIS. An analysis of the impact of multi-threading on communication performance. Dans *CAC 2009 : The 9th Workshop on Communication Architecture for Clusters, held in conjunction with IPDPS 2009*. IEEE Computer Society Press, Rome, Italy, May 2009. doi :10.1109/IPDPS.2009.5160893. `http://hal.inria.fr/inria-00381670`.

[24] J. JOSE, M. LUO, S. SUR et D. K. PANDA. Unifying UPC and MPI Runtimes : Experience with MVAPICH. Dans *Fourth Conference on Partitioned Global Address Space Programming Model (PGAS10)*, October 2010.

[25] K. B. FERREIRA, P. BRIDGES et R. BRIGHTWELL. Characterizing application sensitivity to OS interference using kernel-level noise injection. Dans *SC '08 : Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pp. 1–12. IEEE Press, Piscataway, NJ, USA, 2008. ISBN 978-1-4244-2835-9.

[26] K. FERREIRA, K. T. PEDRETTI, M. LEVENHAGEN et R. BRIGHTWELL. Exploring Memory Management Strategies in Catamount. Dans *Proceedings of the 2008 Cray Users' Group Annual Technical Conference*. Helsinki, Finland, May 2008.

[27] T. LINDHOLM et F. YELLIN. *The Java Virtual Machine Specification (2nd Edition)*. Prentice Hall, 1999.

[28] E. INTERNATIONAL. Standard ECMA-335, Common Language Infrastructure (CLI), 2005.

[29] B. AMEDRO, V. BODNARTCHOUK, D. CAROMEL, C. DELBE, F. HUET et G. TABOADA. Current State of Java for HPC. Technical Report RT-0353, INRIA, 2008. `http://hal.inria.fr/inria-00312039/en/`.

[30] F. J. CORBATÓ, M. MERWIN-DAGGETT et R. C. DALEY. An experimental time-sharing system. Dans *AIEE-IRE '62 (Spring) : Proceedings of the May 1-3, 1962, spring joint computer conference*, pp. 335–344. ACM, New York, NY, USA, 1962. doi :http://doi.acm.org/10.1145/1460833.1460871.

[31] M. VARIAN. VM and the VM community, past present, and future. Dans *SHARE 89 Sessions 9059-9061*, 1997. http://www.princeton.edu/~melinda/25paper.pdf.

[32] J. HOWARD, S. DIGHE, Y. HOSKOTE, S. VANGAL, D. FINAN, G. RUHL, D. JENKINS, H. WILSON, N. BORKAR, G. SCHROM, F. PAILET, S. JAIN, T. JACOB, S. YADA, S. MARELLA, P. SALIHUNDAM, V. ERRAGUNTLA, M. KONOW, M. RIEPEN, G. DROEGE, J. LINDEMANN, M. GRIES, T. APEL, K. HENRISS, T. LUND-LARSEN, S. STEIBL, S. BORKAR, V. DE, R. VAN DER WIJNGAART et T. MATTSON. A 48-Core IA-32 message-passing processor with DVFS in 45nm CMOS. *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pp. 108–109, feb. 2010. ISSN 0193-6530. doi :10.1109/ISSCC.2010.5434077.

[33] E. BUGNION, S. DEVINE, K. GOVIL et M. ROSENBLUM. Disco : running commodity operating systems on scalable multiprocessors. *ACM Trans. Comput. Syst.*, volume 15(4) :pp. 412–447, 1997. ISSN 0734-2071. doi :http://doi.acm.org/10.1145/265924.265930.

[34] R. P. GOLDBERG. Architecture of virtual machines. Dans *Proceedings of the workshop on virtual computer systems*, pp. 74–112. ACM, New York, NY, USA, 1973. doi :http://doi.acm.org/10.1145/800122.803950.

[35] G. J. POPEK et R. P. GOLDBERG. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, volume 17(7) :pp. 412–421, 1974. ISSN 0001-0782. doi : http://doi.acm.org/10.1145/361011.361073.

[36] J. S. ROBIN et C. E. IRVINE. Analysis of the Intel Pentium's ability to support a secure virtual machine monitor. Dans *SSYM'00 : Proceedings of the 9th conference on USENIX Security Symposium*, pp. 10–10. USENIX Association, Berkeley, CA, USA, 2000.

[37] T. ORMANDY et J. TINNES. Virtualisation security and the Intel privilege model. PacSec 2009 Conference, 2009. http://www.cr0.org/paper/jt-to-virtualisation_security.pdf.

[38] A. WHITAKER, M. SHAW et S. D. GRIBBLE. Scale and performance in the Denali isolation kernel. Dans *OSDI '02 : Proceedings of the 5th symposium on Operating systems design and implementation*, pp. 195–209. ACM, New York, NY, USA, 2002. doi :http://doi.acm.org/10.1145/1060289.1060308.

[39] P. BARHAM, B. DRAGOVIC, K. FRASER, S. HAND, T. HARRIS, A. HO, R. NEUGEBAUER, I. PRATT et A. WARFIELD. Xen and the art of virtualization. Dans *SOSP '03 : Proceedings of the nineteenth ACM symposium on Operating systems principles*, pp. 164–177. ACM, New York, NY, USA, 2003. ISBN 1-58113-757-5. doi :http://doi.acm.org/10.1145/945445.945462.

[40] J. LEVASSEUR, V. UHLIG, M. CHAPMAN, P. CHUBB, B. LESLIE et G. HEISER. Pre-virtualization : Slashing the cost of virtualization. Technical Report 2005-30, Fakultät für Informatik, Universität Karlsruhe (TH), November 2005.

[41] R. BHARGAVA, B. SEREBRIN, F. SPADINI et S. MANNE. Accelerating two-dimensional page walks for virtualized systems. Dans *ASPLOS XIII : Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pp. 26–35. ACM, New York, NY, USA, 2008. ISBN 978-1-59593-958-6. doi :http://doi.acm.org/10.1145/1346281.1346286.

[42] Vhost-Net. http://www.linux-kvm.org/page/VhostNet.

[43] PCI-SIG Single Root I/O Virtualization 1.0 Specification. http://www.pcisig.com/specifications/iov/single_root.

[44] J. LIU, W. HUANG, B. ABALI et D. K. PANDA. High performance VMM-bypass I/O in virtual machines. Dans *ATEC '06 : Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, pp. 3–3. USENIX Association, Berkeley, CA, USA, 2006.

[45] Y. ETSION et D. TSAFRIR. A Short Survey of Commercial Cluster Batch Schedulers. Technical Report 2005-13, The Hebrew University, May 2005.

[46] D. A. LIFKA. The ANL/IBM SP Scheduling System. Dans *IPPS '95 : Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pp. 295–303. Springer-Verlag, London, UK, 1995. ISBN 3-540-60153-8.

[47] A. W. MU'ALEM et D. G. FEITELSON. Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling. *IEEE Trans. Parallel and Distributed Systems*, volume 12(6) :pp. 529–543, 2001.

[48] F. HERMENIER, X. LORCA, J.-M. MENAUD, G. MULLER et J. LAWALL. Entropy : a consolidation manager for clusters. Dans *VEE '09 : Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pp. 41–50. ACM, New York, NY, USA, 2009. ISBN 978-1-60558-375-4. doi :http://doi.acm.org/10.1145/1508293.1508300.

[49] L. GRIT, D. IRWIN, A. YUMEREFENDI et J. CHASE. Virtual Machine Hosting for Networked Clusters : Building the Foundations for "Autonomic" Orchestration. Dans *VTDC '06 : Proceedings of the 2nd International Workshop on Virtualization Technology in Distributed Computing*, p. 7. IEEE Computer Society, Washington, DC, USA, 2006. ISBN 0-7695-2873-1. doi :http://dx.doi.org/10.1109/VTDC.2006.17.

[50] N. FALLENBECK, H.-J. PICHT, M. SMITH et B. FREISLEBEN. Xen and the Art of Cluster Scheduling. Dans *VTDC '06 : Proceedings of the 2nd International Workshop on Virtualization Technology in Distributed Computing*, p. 4. IEEE Computer Society, Washington, DC, USA, 2006. ISBN 0-7695-2873-1. doi :http://dx.doi.org/10.1109/VTDC.2006.18.

[51] B. SOTOMAYOR, K. KEAHEY et I. FOSTER. Combining batch execution and leasing using virtual machines. Dans *HPDC '08 : Proceedings of the 17th international symposium on High performance distributed computing*, pp. 87–96. ACM, New York, NY, USA, 2008. ISBN 978-1-59593-997-5. doi :http://doi.acm.org/10.1145/1383422.1383434.

[52] A. VERMA, P. AHUJA et A. NEOGI. Power-aware dynamic placement of HPC applications. Dans *ICS '08 : Proceedings of the 22nd annual international conference on Supercomputing*, pp. 175–184. ACM, New York, NY, USA, 2008. ISBN 978-1-60558-158-3. doi :http://doi.acm.org/10.1145/1375527.1375555.

[53] W. EMENEKER et D. STANZIONE. Increasing Reliability through Dynamic Virtual Clustering. Dans *High Availability and Performance Computing Workshop*, 2006.

[54] C. ENGELMANN, G. R. VALLEE, T. NAUGHTON et S. L. SCOTT. Proactive Fault Tolerance Using Preemptive Migration. Dans *PDP '09 : Proceedings of the 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pp. 252–257. IEEE Computer Society, Washington, DC, USA, 2009. ISBN 978-0-7695-3544-9. doi : http://dx.doi.org/10.1109/PDP.2009.31.

[55] J. LANGE, K. PEDRETTI, T. HUDSON, P. DINDA, Z. CUI, L. XIA, P. BRIDGES, A. GOCKE, S. JACONETTE, M. LEVENHAGEN et R. BRIGHTWELL. Palacios and Kitten : New High Performance Operating Systems for Scalable Virtualized and Native Supercomputing. Dans *IPDPS '10 : Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium*. IEEE Computer Society, Washington, DC, USA, April 2010.

[56] E. VAN HENSBERGEN. P.R.O.S.E. : partitioned reliable operating system environment. *SIGOPS Oper. Syst. Rev.*, volume 40(2) :pp. 12–15, 2006. ISSN 0163-5980. doi :http://doi.acm.org/10.1145/1131322.1131329.

[57] M. BUTRICO, D. DA SILVA, O. KRIEGER, M. OSTROWSKI, B. ROSENBURG, D. TSAFRIR, E. VAN HENSBERGEN, R. W. WISNIEWSKI et J. XENIDIS. Specialized execution environ-

ments. *SIGOPS Oper. Syst. Rev.*, volume 42(1) :pp. 106–107, 2008. ISSN 0163-5980. doi : http://doi.acm.org/10.1145/1341312.1341335.

[58] S. THIBAULT et T. DEEGAN. Improving Performance by Embedding HPC Applications in Lightweight Xen Domains. Dans *2nd Workshop on System-level Virtualization for High Performance Computing (HPCVIRT'08)*. Glasgow, Scotland, March 2008. `http://hal.inria.fr/inria-00329969`.

[59] G. AMMONS, J. APPAVOO, M. BUTRICO, D. DA SILVA, D. GROVE, K. KAWACHIYA, O. KRIEGER, B. ROSENBURG, E. VAN HENSBERGEN et R. W. WISNIEWSKI. Libra : a library operating system for a jvm in a virtualized execution environment. Dans *VEE '07 : Proceedings of the 3rd international conference on Virtual execution environments*, pp. 44–54. ACM, New York, NY, USA, 2007. ISBN 978-1-59593-630-1. doi :http://doi.acm.org/10.1145/1254810.1254817.

[60] M. JORDAN. JavaGuest - A Research Java Virtual Machine on Xen. Dans *Xen Summit*, November 2007.

[61] G. W. DUNLAP, D. G. LUCCHETTI, M. A. FETTERMAN et P. M. CHEN. Execution replay of multiprocessor virtual machines. Dans *VEE '08 : Proceedings of the fourth ACM SIGPLAN/-SIGOPS international conference on Virtual execution environments*, pp. 121–130. ACM, New York, NY, USA, 2008. ISBN 978-1-59593-796-4. doi :http://doi.acm.org/10.1145/1346256.1346273.

[62] M. XU, V. MALYUGIN, J. SHELDON, G. VENKITACHALAM et B. WEISSMAN. Retrace : Collecting execution trace with virtual machine deterministic replay. Dans *MoBS '07 : Proceedings of the 3rd Annual Workshop on Modeling, Benchmarking and Simulation*, 2007.

[63] S. T. KING, G. W. DUNLAP et P. M. CHEN. Debugging operating systems with time-traveling virtual machines. Dans *ATEC '05 : Proceedings of the annual conference on USENIX Annual Technical Conference*, pp. 1–1. USENIX Association, Berkeley, CA, USA, 2005.

[64] R. G. MINNICH et D. W. RUDISH. Ten Million and One Penguins, or, Lessons Learned from booting millions of virtual machines on HPC systems. Dans *HPCVirt '10 : 4th Workshop on System-level Virtualization for High Performance Computing*. ACM, New York, NY, USA, April 2010.

[65] M. CHAPMAN et G. HEISER. vNuma : A virtual Shared Memory Multiprocessor. Dans *Proceedings of the 2009 USENIX Annual Technical Conference*, 2009.

[66] Versatile SMP (vSMP) Architecture. `http://www.scalemp.com/architecture`.

[67] J. PENG, X. LONG et L. XIAO. Providing Virtualization-Based Single System Image on Clusters. Dans *GCC '08 : Proceedings of the 2008 Seventh International Conference on Grid and Cooperative Computing*, pp. 28–32. IEEE Computer Society, Washington, DC, USA, 2008. ISBN 978-0-7695-3449-7. doi :http://dx.doi.org/10.1109/GCC.2008.121.

[68] A. RANADIVE, M. KESAVAN, A. GAVRILOVSKA et K. SCHWAN. Performance Implications of Virtualizing Multicore Cluster Machines. Dans *2nd Workshop on System-level Virtualization for High Performance Computing (HPCVirt'08)*, 2008.

[69] V. UHLIG, J. LEVASSEUR, E. SKOGLUND et U. DANNOWSKI. Towards Scalable Multiprocessor Virtual Machines. Dans *Proceedings of the 3rd Virtual Machine Research and Technology Symposium (VM'04)*, 2004. `http://l4ka.org/publications/`.

[70] H. A. LAGAR-CAVILLA, J. WHITNEY, A. SCANNELL, P. PATCHIN, S. M. RUMBLE, E. DE LARA, M. BRUDNO et M. SATYANARAYANAN. SnowFlock : Rapid Virtual Machine Cloning for Cloud Computing. Dans *3rd European Conference on Computer Systems (Eurosys)*. Nuremberg, Germany, April 2009.

[71] L. CHAI, A. HARTONO et D. K. PANDA. Designing High Performance and Scalable MPI Intra-node Communication Support for Clusters. Dans *Proceedings of the 2006 IEEE International Conference on Cluster Computing (Cluster'06)*, 2006. ISSN 1552-5244. doi : 10.1109/CLUSTR.2006.311850.

[72] D. BUNTINAS, G. MERCIER et W. GROPP. Implementation and Shared-Memory Evaluation of MPICH2 over the Nemesis Communication Subsystem. Dans *Proceedings of the 13th European PVM/MPI Users Group Meeting (EuroPVM/MPI'06)*, 2006.

[73] H.-W. JIN et D. K. PANDA. LiMIC : Support for High-Performance MPI Intra-node Communication on Linux Cluster. Dans *Proceedings of the 2005 International Conference on Parallel Processing (ICPP'05)*, 2005. doi :http://dx.doi.org/10.1109/ICPP.2005.48.

[74] D. BUNTINAS, B. GOGLIN, D. GOODELL, G. MERCIER et S. MOREAUD. Cache-Efficient, Intranode Large-Message MPI Communication with MPICH2-Nemesis. Dans *Proceedings of the 38th International Conference on Parallel Processing (ICPP-2009)*, pp. 462–469. IEEE Computer Society Press, Vienna, Austria, September 2009. doi :10.1109/ICPP.2009.22. `http://hal.inria.fr/inria-00390064`.

[75] E. D. DEMAINE. A Threads-Only MPI Implementation for the Development of Parallel Programs. Dans *Proceedings of the 11th International Symposium on High Performance Computing Systems (HPCS'97)*, 1997.

[76] X. ZHANG, S. MCINTOSH, P. ROHATGI et J. L. GRIFFIN. XenSocket : A High-Throughput Interdomain Transport for Virtual Machines. Dans *Proceedings of the 8th ACM/IFIP/USENIX International Middleware Conference (Middleware 2007)*, 2007.

[77] K. KIM, C. KIM, S.-I. JUNG et H.-S. SHIN. Inter-domain socket communications supporting high performance and full binary compatibility on Xen. Dans *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments (VEE'08)*, 2008.

[78] W. HUANG, M. KOOP, Q. GAO et D. K. PANDA. Virtual Machine Aware Communication Libraries for High Performance Computing. Dans *Proceedings of the 2007 ACM/IEEE conference on Supercomputing (SC'07)*, 2007.

[79] D. BONACHEA et J. DUELL. Problems with using MPI 1.1 and 2.0 as compilation targets for parallel language implementations. *International Journal of High Performance Computing and Networking*, volume 1(1-3) :pp. 91–99, 2004. ISSN 1740-0562. doi :http://dx.doi.org/10.1504/IJHPCN.2004.007569.

[80] R. THAKUR et W. GROPP. Improving the Performance of Collective Operations in MPICH. Dans *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 2840 of *Lecture Notes in Computer Science*, pp. 257–267. Springer Berlin / Heidelberg, 2003.

[81] G. MERCIER et J. CLET-ORTEGA. Towards an Efficient Process Placement Policy for MPI Applications in Multicore Environments. Dans *EuroPVM/MPI*, volume 5759 of *Lecture Notes in Computer Science*, pp. 104–115. Springer, Espoo, Finland, September 2009. doi : 10.1007/978-3-642-03770-2_17. `http://hal.inria.fr/inria-00392581`.

[82] H. JOURDREN. HERA : A Hydrodynamic AMR Platform for Multi-Physics Simulations. Dans T. PLEWA, T. LINDE et V. GREGORY WEIRS, editors, *Adaptive Mesh Refinement - Theory and Applications*, volume 41 of *Lecture Notes in Computational Science and Engineering*, pp. 283–294. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-27039-3.

[83] G. MERCIER et J. CLET-ORTEGA. Towards an Efficient Process Placement Policy for MPI Applications in Multicore Environments. Dans *EuroPVM/MPI*, volume 5759 of *Lecture Notes in Computer Science*, pp. 104–115. Springer, Espoo, Finland, September 2009. doi : 10.1007/978-3-642-03770-2_17. `http://hal.inria.fr/inria-00392581`.

Annexe A

Fonctions MPI implémentées

```
int MPI_Init(int * argc, char *** argv );
int MPI_Init_thread(int * argc, char *** argv, int required, int *provided);
int MPI_Initialized(int * flags);
int MPI_Finalize(void);

int MPI_Comm_rank(MPI_Comm comm, int * rank);
int MPI_Comm_size(MPI_Comm comm, int * size);
int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm * newcomm);
int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm);

int MPI_Isend(void * buf, int count, MPI_Datatype datatype, int dest,
              int tag, MPI_Comm comm, MPI_Request * request);
int MPI_Irecv(void * buf, int count, MPI_Datatype datatype, int source,
              int tag, MPI_Comm comm, MPI_Request * request);
int MPI_Wait(MPI_Request * request, MPI_Status * status);
int MPI_Waitall(int count, MPI_Request * request, MPI_Status * status);
int MPI_Waitsome(int incount, MPI_Request * requests, int * outcount,
                 int * indices, MPI_Status * statuses);

int MPI_Send(void * buf, int count, MPI_Datatype datatype, int dest, int tag,
             MPI_Comm comm);
int MPI_Recv(void * buf, int count, MPI_Datatype datatype, int source, int tag,
             MPI_Comm comm, MPI_Status * status);
int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype, int dest,
                 int sendtag, void * recvbuf, int recvcount,
                 MPI_Datatype recvtype, int source, int recvtag, MPI_Comm comm,
                 MPI_Status * status);

int MPI_Barrier(MPI_Comm comm);
int MPI_Alltoall(void * sendbuf, int sendcount, MPI_Datatype sendtype,
                 void * recvbuf,int recvcount, MPI_Datatype recvtype,
                 MPI_Comm comm);
int MPI_Alltoallv(void * sendbuf, int * sendcount, int * sdispls,
                  MPI_Datatype sendtype, void * recvbuf, int * recvcount,
                  int * rdispls, MPI_Datatype recvtype, MPI_Comm comm);

int MPI_Allgatherv(void * sendbuf, int sendcount, MPI_Datatype sendtype,
                   void * recvbuf, int * recvcounts, int * displs,
                   MPI_Datatype recvtype, MPI_Comm comm);
int MPI_Allgather(void * sendbuf, int sendcount, MPI_Datatype sendtype,
                  void * recvbuf, int recvcount, MPI_Datatype recvtype,
                  MPI_Comm comm);
int MPI_Gather(void * sendbuf, int sendcount, MPI_Datatype sendtype,
```

```
                void * recvbuf, int recvcount, MPI_Datatype recvtype,
                int root, MPI_Comm comm);
int MPI_Gatherv(void * sendbuf, int sendcount, MPI_Datatype sendtype,
                void * recvbuf, int * recvcounts, int * displs,
                MPI_Datatype recvtype, int root, MPI_Comm comm);

int MPI_Scatter(void * sendbuf, int sendcount, MPI_Datatype sendtype,
                void * recvbuf, int recvcount, MPI_Datatype recvtype,
                int root, MPI_Comm comm);

int MPI_Scatterv(void * sendbuf, int * sendcounts, int * displs,
                 MPI_Datatype sendtype, void * recvbuf, int recvcount,
                 MPI_Datatype recvtype, int root, MPI_Comm comm);

int MPI_Reduce(void * sendbuf, void * recvbuf, int count, MPI_Datatype datatype,
               MPI_Op op, int root, MPI_Comm comm);
int MPI_Allreduce(void* sendbuf, void * recvbuf, int count,
                  MPI_Datatype datatype, MPI_Op op, MPI_Comm comm);

int MPI_Bcast(void * buffer, int count, MPI_Datatype datatype, int root,
              MPI_Comm comm);


int MPI_Reduce_scatter(void *sendbuf, void *recvbuf, int *recvcounts,
                       MPI_Datatype datatype, MPI_Op op, MPI_Comm comm);

int MPI_Type_vector(int count, int blocklength, int stride,
                    MPI_Datatype oldtype, MPI_Datatype *newtype);
int MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype);

int MPI_Type_size(MPI_Datatype datatype, int * size);

int MPI_Type_struct(int count, int * blocklens, MPI_Aint * indices,
                    MPI_Datatype * old_types, MPI_Datatype * newtype);

int MPI_Type_commit(MPI_Datatype *datatype);

int MPI_Type_free(MPI_Datatype * datatype);

int MPI_Error_string(int errorcode, char * string, int * resultlen);

double MPI_Wtime();

int MPI_Get_version(int * version, int * subversion);

int MPI_Pack(void * inbuf, int incount, MPI_Datatype datatype, void * outbuf,
             int outsize, int * position, MPI_Comm comm);

int MPI_Unpack(void * inbuf, int insize, int * position, void * outbuf,
               int outcount, MPI_Datatype datatype, MPI_Comm comm);

int MPI_Pack_size(int incount, MPI_Datatype datatype, MPI_Comm comm,
                  int * size);

typedef void (MPI_User_function) (void * invec, void * outvec, int * len,
                                  MPI_Datatype * datatype);
int MPI_Op_create(MPI_User_function * function, int commute, MPI_Op * op);
int MPI_Op_free(MPI_Op * Op);
```